МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

М. І. Безменов

ТУРБО ПАСКАЛЬ 7.0

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів, які навчаються за напрямами «Прикладна математика» та «Інформатика»

Харків 2006

ББК 018.1
Б 39
УДК 004.432.2

Р е ц е н з е н т и

А. І. Пушкар, д-р економ. наук, проф., завідувач кафедри обчислювальної техніки і програмування Харківського державного економічного університету,

Е. В. Бодянський, д-р техн. наук, проф. кафедри штучного інтелекту Харківського національного університету радіоелектроніки

Гриф наданий Міністерством освіти і науки України,
лист № 14/18.2-258 від 18.02.04

Безменов М. І.

Б 39 Турбо Паскаль 7.0: Навч. посібник. – Харків: НТУ «ХПІ»; Парус™, 2005.– 240 с.

ISBN 966-593-389-2

У навчальному посібнику описані можливості однієї з найпопулярніших мов програмування — Turbo Pascal (версія 7.0). Наведено досить повне описання мови, а також її основних модулів (Crt, Dos, Graph, Strings). Викладення супроводжується великою кількістю прикладів програм.

Призначено для студентів, які навчаються за напрямами «Прикладна математика» та «Інформатика», а також для студентів інших технічних та економічних спеціальностей. Може рекомендуватися учням випускних класів середніх учбових закладів.

Табл. 4. Бібліогр. 6 назв.

ББК 018.1

ISBN 966-593-389-2

ЗМІСТ

ВСТУП

На зорі розвитку електронної обчислювальної техніки програми складалися в машинних кодах, що висувало особливі вимоги до кваліфікації програмістів і обумовлювало труднощі в написанні й сприйнятті вже написаних програм. Поява алгоритмічних мов програмування поступово зняла ореол таємничості з процесу програмування і зробила його доступним і зрозумілим для широкого кола користувачів. Великого поширення набули такі мови, як FORTRAN, Algol-60, PL/1, Basic. З розвитком обчислювальної техніки необхідність у залученні до процесу написання програм широкого кола фахівців обумовила розробку мови «для навчання програмуванню як систематичній дисципліні». Як така мова професором Федерального технологічного інституту в Цюріху Н. Віртом була запропонована мова Pascal, що створювалася в 1968–1971 роках.

Розвиток і удосконалювання мови призвели до розробки в 1979 році її стандартної версії, а також великої кількості діалектів і прикладних бібліотек.

Істотний внесок у поширення мови серед самого широкого кола користувачів забезпечила фірма Borland International, яка розширила стандарт мови Pascal і озброїла його потужним компілятором, фактично створивши нову версію мови, що одержала назву Turbo Pascal (TP). У 1985 р. ця фірма запропонувала свою версію мови для персональних ЕОМ (версія TP 3.0). Розроблена пізніше версія TP 4.0 характерна тим, що в ній мова програмування була з’єднана з текстовим редактором. Надалі мова удосконалювалася відповідно до вимог поточного моменту і тенденціям розвитку програмування. До останнього часу мова програмування TP 7.0 залишалася однією з найбільш розповсюджених мов, поєднуючи в собі великі можливості й зручність інтерфейсу користувача. Її основні конструкції включили до себе такі мови, як Object Pascal і Delphi.

Дана робота ставить за мету дати уявлення про основні можливості мови TP 7.0 і, будучи розширенням конспекту лекцій, може бути корисною, на думку автора, усім бажаючим освоїти цю мову програмування.

1. БАЗОВІ ПОНЯТТЯ МОВИ

1.1. Алфавіт мови

Основні символи мови ТР – літери, арабські цифри і спеціальні символи – складають його алфавіт.

У ТР використовуються прописні і малі літери латинського алфавіту; у деяких випадках можуть бути використані літери українського алфавіту.

Спеціальні символи містять у собі:

а) знаки операцій + – * / = < > <> <= >= :=

б) обмежники . , ( ) [ ] { } : ; ’ ”

в) спеціальні знаки $ @ # & ^ _ ~

Застосовуються, крім того, символ «прогалина», а також всі інші символи, що є на клавіатурі, і символи, що не друкуються.

У програмі на ТР повинно бути забезпечено використання комірок пам’яті, для чого служать змінні.

Змінна — це ім’я фізичної ділянки пам’яті, в якій у будь-який момент часу може зберігатися тільки одне значення. Для розрізнення змінних використовуються імена (ідентифікатори).

Ім’я (змінної, програми чи якогось програмного об’єкта) — це послідовність припустимих у ТР символів, що задовольняє таким правилам:

- ім’я може складатися тільки з латинських літер, цифр і символів підкреслення (при цьому великі і малі літери не розрізняються);
- ім’я повинне починатися тільки з латинської букви чи символу підкреслення «_».

Довжина імені не може перевищувати довжини рядка (127 символів), але розрізняються імена тільки за першими 63 символами.

Слова, уживані в ТР, підрозділяються на три групи: службові слова; визначені імена; імена, обумовлені програмістом.

Службові (*резервовані*) *слова* — це слова мови, які мають спеціальне, раз і назавжди закріплене за ними значення. Їх не можна використовувати ніяк інакше (наприклад, не можна позначити змінну ім’ям **end**).

Визначені (*стандартні*) *імена* також мають спеціальне (заздалегідь задане) значення. Однак програміст може обходити

закріплені за ними значення і використовувати їх як імена, визначені програмістом. Якщо програміст не визначить явно, з якою метою застосовується те чи інше стандартне ім'я в програмі, воно буде сприйматися у властивому даному імені визначеному значенні (наприклад, визначеними є імена **Integer**, **WriteLn**).

Всі інші імена є іменами, визначеними програмістом, чи інакше *користувальницькими іменами*. Вони повинні бути вичерпно оголошені в програмі.

Ще одним елементом мови є явно задані константи і насамперед числа.

Десяткові числа завжди починаються з цифри, перед якою може стояти знак числа (+ чи –). Дійсні числа зображуються в двох форматах. У форматі з фіксованою крапкою явно вказують положення десяткової крапки (наприклад, 1.002, –12.34, +1.0). У форматі з плавучою крапкою використовується десятковий порядок, що позначається прописною чи малою літерою E, слідом за якою йде ціле число, що вказує значення порядку, наприклад, 15e14, +12.6e–1, 1.0E+03.

У TP максимальне ціле число — це 2147483647 (воно міститься у визначеній цілій константі **MaxLongInt**). Діапазон цілих чисел — від –2147483648 до 2147483647. Визначена в TP ціла константа **MaxInt** містить у собі значення 32767. Компілятор TP дозволяє оперувати з 38-м порядком у поданні дійсних чисел, а деякі компілятори – до 67-го.

Ціле число може задаватися не тільки в десятковій, але й у шістнадцятковій системі числення. Шістнадцяткове число складається з шістнадцяткових цифр (0, 1, 2, ..., 9, A, B, C, D, E, F), яким передує знак долара $. Діапазон шістнадцяткових чисел – від **$00000000** до **$FFFFFFFF**.

Текстовим літералом (стрингом, рядком) у мові TP називають послідовність будь-яких припустимих символів, що розташовані між апострофами (наприклад, 'Турбо Паскаль'). Якщо як символ рядка необхідно використовувати апостроф, то записують підряд два апострофи. Рядок можна задати також у вигляді послідовності, утвореної із символів # з наступним цифровим кодом (наприклад, запис #88#89#90 еквівалентний рядку 'XYZ') чи комбінуванням таких послідовностей і рядків в апострофах (наприклад, #219'Turbo'#219'Pascal'#219). У рядкових даних великі і малі літери розрізняються.

1.2. Структура програми. Оголошення змінних

Програма на ТР записується у вигляді послідовності тверджень (операторів), які можуть записуватися в один чи кілька рядків. Вона складається з заголовка програми (який може бути опущений) і тіла програми (блоку).

Заголовок програми задається службовим словом **program**, слідом за яким йде ім'я програми.

Тіло програми складається з двох частин:

- розділу оголошення даних;
- здійсненної частини.

У *розділі оголошень* задаються імена даних, визначаються їхні типи, можливі значення й області пам'яті, необхідні для розміщення даних. Типи даних бувають простими і складними, вони можуть задаватися програмістом або належати до одного зі стандартних типів.

Здійсненна частина програми починається зі службового слова **begin** і завершується службовим словом **end** з крапкою (**end.**), що означає кінець тексту програми.

Оператори в програмі відокремлюються один від одного символом «;» (цей символ не є частиною оператора). Досить часто декілька операторів необхідно виконати спільно. У цьому випадку вони облямовуються службовими словами **begin** і **end**. Послідовність операторів, яка відкривається службовим словом **begin** і завершується службовим словом **end**, є *складеним оператором*.

У деяких випадках виникає необхідність у використанні так званого *порожнього оператора*, тобто оператора, що не виконує ніяких дій. Він позначається відсутністю оператора перед крапкою з комою між службовими словами **then** і **else**.

У програмі можуть використовуватися коментарі. *Коментар* — це текст, розміщений між символами **{** і **}** чи між парами символів **(*** і ***)**. Він розташовується в будь-якому місці, де може стояти прогалина. Взагалі говорячи, прогалини служать для поділу елементів програми (імен, констант і т.д.). Замість однієї прогалини можна поставити будь-яку їхню кількість. Прогалини всередині імен, констант, складених символів не допускаються. Перехід до нового рядка дозволяється у будь-якому місці, де може стояти прогалина. Рекомендується використовувати прогалини, коментарі й перехід до нового рядка для виділення окремих змістових частин програми і вкладених конструкцій.

Якщо який-небудь коментар починається із символу $, то він називається *директивою компіляції* і встановлює режим, при якому повинна здійснюватися компіляція програми. У ТР існує три види директив компіляції: директиви-перемикачі, директиви з параметрами й умовні директиви. Усі вони оформляються у вигляді рядка коментарів, у якому першим символом (без прогалин) є символ $ (наприклад, **{$I+}**). *Директиви-перемикачі* вказують на те, що компілятор повинен виконати чи не виконати деякі дії. Активізація директиви позначається символом «+ », вимикання – символом «-» слідом за ім'ям директиви. Директиви-перемикачі можна згадувати роздільно чи спільно. В останньому випадку вони розділяються комами, а символ $ ставиться один раз **{$R+,I–,N+}**. Якщо при завданні директив-перемикачів десь буде поставлена прогалина, то весь інший текст у фігурних дужках сприймається як коментар. У *директивах з параметрами* вказуються параметри (ім'я файла, розміри пам'яті і т.д.), які передаються компілятору для компіляції. *Умовні директиви* використовуються при налагодженні для перевірки окремих фрагментів програми.

Щоб можна було використовувати яке-небудь ім'я в програмі, його потрібно оголосити до першого застосування. Те саме ім'я не може бути оголошене двічі усередині однієї програмної одиниці.

Для оголошення змінних служить спеціальний розділ (їх може бути декілька), що починається зі службового слова **var**, за яким йдуть оператори оголошення окремих змінних. Для оголошення змінної вказуються її ім'я і (після двокрапки) тип змінної. При оголошенні декількох змінних одного типу імена можуть бути перелічені через кому в одному операторі:

```
var
    number,i,j: Integer;
```

Розділ оголошення змінних подає інформацію для розподілу пам'яті під використовувані в програмі змінні.

Найпростішими стандартними типами даних є **Integer** (цілий), **Real** (дійсний), **Boolean** (булевський), **Char** (символьний, літерний).

Тип визначає можливі значення змінних, констант, функцій, виразів, які належать даному типу, форму подання їх у машині, обсяг пам'яті, що вони займають, й операції, що можуть виконуватися над ними.

1.3. Система типів

Система типів мови ТР досить велика. Базовими є *прості* типи; складені типи за певними правилами будуються з простих. Ці типи називають також *скалярними*. Стандартні прості типи поділяються на чотири групи: *цілі, дійсні, символьний, булевські*. На основі стандартних скалярних типів можна утворювати користувальницькі скалярні типи. До скалярних відносять також *перелічувані типи*, що оголошуються користувачем. Будь-який скалярний тип характеризується його можливими значеннями, серед яких встановлений лінійний порядок. Усі скалярні типи, крім дійсних, називаються також *порядковими* (дискретними), оскільки для кожного з їхніх значень є попереднє і наступне значення. Обмежені типи (*відрізки*) формуються з порядкових типів шляхом звуження області припустимих значень.

Крім простих, у ТР є *складені* типи (масиви, рядки, записи, множини, файли, об'єкти), формування яких засноване на використанні інших типів. З будь-яких типів можуть бути утворені *посилальні* типи. В особливу групу можна виділити *процедурні* типи. На основі будь-яких типів (у тому числі перелічуваних, обмежених, складених, посилальних і процедурних) можна утворити користувальницькі типи.

1.4. Байт. Слово

Усі дані в комп'ютері подаються в двійковій системі числення. Мінімальною одиницею пам'яті, що може бути адресованою, є байт, який складається з 8 двійкових розрядів – бітів, які нумеруються від нуля. Два розташованих поруч байти, молодший з яких має парну адресу, називаються словом. У байті й у слові найправіший біт є молодшим (номер 0), найлівіший – старшим (з номером 7 чи 15). При записі чисел зі знаком старший біт є знаковим.

1.5. Оператор призначення

Найпростішим способом надання змінній значення є виконання твердження (оператора) *призначення* (*присвоювання*), що має такий вигляд:

змінна:=вираз;

Порядок дії: спочатку обчислюється значення виразу, після чого результат записується в змінну:

```
i:=1;                    {i=1}
number:=i+1;             {number=2}
```

При виконанні оператора призначення треба дотримуватися вимоги сумісності за присвоюванням типів лівої і правої його частин (див. п.1.12.2).

1.6. Стандартні скалярні типи

Кожна з груп стандартних скалярних типів (крім символьних даних) розбивається на кілька підгруп відповідно до того, який обсяг пам'яті виділяється для збереження даних цього типу.

1.6.1. Цілі типи

У TP нараховується 5 цілих типів:

Тип	Діапазон	Пам'ять
ShortInt	–128..127	1 байт
Integer	–32768..32767	2 байт
LongInt	–2147483648..2147483647	4 байт
Byte	0..255	1 байт
Word	0..65535	2 байт

Над даними цілих типів допустимі такі арифметичні операції, що дають цілий результат:

\+ – додавання;

– – віднімання;

* – множення;

div – цілочислове ділення (з відкиданням залишку);

mod – ділення за модулем (виділення залишку від ділення).

Операції відношення, застосовані до цілих операндів, дають результат, який відноситься до типу **Boolean**: **True** чи **False** («істина» чи «неправда»). Операціями відносини (порівняння) є:

= – рівність; <> – нерівність;

>= – більше чи дорівнює; <= – менше чи дорівнює;

> – більше; < – менше.

До аргументів цілого типу можна застосовувати ряд стандартних функцій:

◊ що дають цілий результат:

- **Abs**(*x*) – модуль *x*;
- **Sqr**(*x*) – квадрат *x*;
- **Succ**(*x*) – наступне значення *x* (тобто *x*+1);
- **Pred**(*x*) – попереднє значення *x* (тобто *x*–1);
- **Random**(*x*) – випадкове ціле число з інтервалу 0..*x*-1 (аргумент має тип **Word**; при *x*=0 **Random**(*x*)=0);

◊ що дають дійсний результат:

- **Sin**(*x*) – синус *x*;
- **Cos**(*x*) – косинус *x*;
- **ArcTan**(*x*) – арктангенс *x*;
- **Ln**(*x*) – натуральний логарифм *x*;
- **Exp**(*x*) – експонента *x*;
- **Sqrt**(*x*) – квадратний корінь з *x*;

◊ що дає булевський результат:

- **Odd**(*x*) – **True**, якщо *x* – непарне ціле число, **False** – в іншому випадку.

Функція **Random** повертає рівномірно розподілене випадкове ціле число при цілому аргументі. Її особливістю є те, що при повторному запуску програми вона повертає ті ж значення. Щоб уникнути цього треба на початку програми викликати процедуру **Randomize**.

Процедури **Inc** і **Dec** можуть мати по одному чи по два параметри цілого типу. Якщо параметрів два (розділяються комою), то значення першого параметра збільшується (для **Inc**) або зменшується (для **Dec**) на величину, що дорівнює значенню другого параметра (наприклад, **Inc**(*x*,2)). Якщо параметр один, то його значення збільшується (у **Inc**) чи зменшується (у **Dec**) на 1 (наприклад, **Dec**(x)).

При запису значень у змінні цілого типу контроль виходу за межу припустимого діапазону не виконується (наприклад, якщо *n* – змінна типу **Integer**, то в другому з наступних двох операторів *n*:=20000;*n*:=2**n* не буде виявлена помилка не тільки під час компіляції, але і при виконанні програми). Тому під час налагодження програми рекомендується вмикати директиву **$R**, встановлюючи її в активний стан **{$R+}**.

Неприємною особливістю дій над цілими даними є те, що в цьому випадку результат виконання арифметичних операцій не контролюється на переповнення. У зв'язку з цим можлива поява помилок, що

важко виявляються. Наприклад, якщо *n* – змінна типу Integer, то результатом виконання операторів *n*:=10000; *n*:=*n**10000 **div** 10000 є значення 0, і ця помилка не буде виявлена, навіть якщо буде активізована директива **$R**. У TP 7.0 для контролю переповнення при виконанні операцій над цілими введена директива **$Q**, що за замовчуванням знаходиться в стані **{$Q-}**. При необхідності включення контролю переповнення ця директива повинна бути в потрібному місці активізована: **{$Q+}**.

1.6.2. Дійсні типи

Змінні для збереження даних, які реалізуються дійсними числами, найчастіше визначають з типом **Real**. Однак при необхідності в програмах можна використовувати й інші дійсні типи. Усього в TP визначено 5 дійсних типів:

Тип	Діапазон	Кількість цифр	Пам’ять
Real	2.9e–39 .. 1.7e38	11–12	6 байт
Single	1.5e–45 .. 3.4e38	7–8	4 байт
Double	5.0e–324 .. 1.7e308	15–16	8 байт
Extended	3.4e–4932 .. 1.1e493	19–20	10 байт
Comp	–9.2e63 .. (9.2e63)–1	19–20	8 байт

Тип **Comp** містить тільки цілі значення, що подаються в обчисленнях як дійсні. Усі дійсні типи, крім **Real**, можна використовувати тільки при наявності в конфігурації комп’ютера математичного співпроцесора чи при підключенні засобів його програмної емуляції. Якщо в конфігурації комп’ютера є математичний співпроцесор, то для забезпечення можливості оперування всіма дійсними типами необхідно активізувати директиву компіляції **$N**, задавши її у вигляді **{$N+}**. За відсутністю математичного співпроцесора можна забезпечити його програмну емуляцію, для чого разом з директивою **$N** слід активізувати директиву **$E**: **{$N+,E+}**.

Над дійсними операндами виконуються такі операції, що дають дійсний результат:

+ – додавання; **–** – віднімання; ***** – множення; **/** – ділення,

а також операції відношення (порівняння).

Якщо застосувати операцію ділення до двох цілих операндів, то результат буде дійсним.

До дійсних аргументів можуть бути застосовані раніше розглянуті функції **Abs, Sqr, Sqrt, Sin, Cos, ArcTan, Ln, Exp,** а також функції **Int** (повертає у вигляді дійсного значення цілу частина аргументу) і **Frac** (повертає дробову частину аргументу). Дві функції дають цілий результат:

Trunc(*x*) – відсікання дробової частини;

Round(*x*) – округлення до найближчого цілого.

Функція **Random** без аргументу повертає рівномірно розподілене випадкове число r з інтервалу $0<r\leq 1$. Функція **Pi**, що не має аргументу, повертає число Піфагора 3.1415...

1.6.3. Булевські типи

Тип **Boolean** визначає ті дані, які можуть приймати логічні значення **True** чи **False**. Їхні значення займають 1 байт. До булевських змінних застосовуються такі логічні операції:

not – заперечення;

and – кон'юнкція («і»);

or – диз'юнкція («чи»);

xor – виключаюче «чи».

Якщо *a* і *b* – змінні типу **Boolean**, то результат виконання логічних операцій над ними задається такою таблицею:

a	*b*	**not** *a*	*a* **and** *b*	*a* **or** b	*a* **xor** *b*
False	**False**	**True**	**False**	**False**	**False**
False	**True**	**True**	**False**	**True**	**True**
True	**False**	**False**	**False**	**True**	**True**
True	**True**	**False**	**True**	**True**	**False**

Крім типу **Boolean**, у TP 7.0 введені ще три булевських типи – **ByteBool** (займає 1 байт), **WordBool** (займає 2 байти) і **LongBool** (займає 4 байти).

Для всіх булевських типів значенню **False** відповідає 0, записаний у виділену під булевську змінну область пам'яті, а значенню **True** – будь-яке ненульове значення. Відмінність нових булевських типів від типу **Boolean** у наступному: при присвоюванні їм значень у виділені під них області пам'яті завжди записується 0 чи 1; у тип **Boolean** при

присвоюванні можуть бути записані й інші значення (див. приклад 1.1).

Складні булевські вирази можуть не оброблятися до кінця, якщо продовження обчислень не змінить результат. За замовчуванням обробка складних булевських виразів виробляється саме так. Якщо булевський вираз в обов'язковому порядку потрібно обробляти до кінця, то це забезпечується вмиканням директиви компіляції **{$B+}**. При завданні **{$B-}** булевського виразу може до кінця не оброблятися.

У ТР **False**<**True**, тому до даних булевського типу можна застосовувати всі операції відносини. Крім того, до них можна застосовувати функції **Ord**, **Succ**, **Pred** і процедури **Inc** і **Dec**. Відмінність у роботі оператора присвоювання при оперуванні новими булевськими типами і типом **Boolean** ілюструє наступна програма.

```
{Приклад 1.1}
{Робота оператора присвоювання і функцій Ord, Pred і Succ
 при оперуванні булевськими типами}
var
  b:Boolean;
  wb:WordBool;
begin
  b:=False;b:=Pred(b);
  WriteLn(b,' ',Ord(b));                          {Друк: TRUE -1}
  WriteLn(b<False);{Друкується TRUE, незважаючи на те, що ...}
  wb:=False;b:=Pred(wb);  {b=True; тут Ord(False)=0,Ord(b)=-1}
  WriteLn(wb,' ',Ord(wb));                        {Друк: FALSE 0}
  b:=True;b:=Succ(b);
  WriteLn(b,' ',Ord(b));                           {Друк: TRUE 2}
  wb:=True;b:=Succ(wb);
  WriteLn(wb,' ',Ord(wb));                         {Друк: TRUE 1}
end.
```

Якщо в даній програмі замінити тип **WordBool** на **ByteBool** чи **LongBool**, то результат її роботи не зміниться.

1.6.4. Символьний тип

Тип **Char** визначає упорядковану сукупність основних символів мови. Цей тип займає 1 байт і служить для подання 256 різних символів, які відрізняються кодом ASCII (американський стандартний код для обміну інформацією), що власне і зберігається у відведеному байті. Символи упорядковані відповідно до їхнього коду (значення кодів від 0 до 255), у зв'язку з чим до даних символьного типу можна застосовувати всі операції відношення. Значення символьної змінної –

один символ. У програмі значення символьного типу можна навести записом, що складається із символу «#» і цілого коду (#214, #17). Якщо символ має екранне подання, то його можна вказати в явному вигляді, уклавши в апострофи (наприклад, ’a’, ’A ’, ’6’, ’–’). Деякі керуючі символи можна навести у вигляді ^C, де C – умовна позначка керуючого символу (наприклад, ^G – символ з кодом 7).

Функція **Ord**(*x*), де *x* – значення символьного типу, повертає код зазначеного символу. Зворотною їй є функція **Chr**(*x*), що дає символьне подання цілого невід’ємного числа *x*.

До даних типу **Char** застосовні функції **Pred**(*x*) і **Succ**(*x*), а також процедури **Inc** і **Dec**.

У складних виразах порядок операцій визначається їхнім пріоритетом. Операції одного пріоритетного рівня виконуються зліва направо. Порядок операцій можна змінити, скориставшись круглими дужками. Значення функцій обчислюються раніше, ніж виконуються інші операції. Пріоритетні рівні розглянутих операцій такі (за убуванням пріоритетів):

1 – одномісний мінус, **not**;
2 – *, /, **div**, **mod**, **and**;
3 – +, -, **or**, **xor**;
4 – <, <=, >, >=, =, <>.

Щоб визначити обсяг пам’яті, що виділяється під елемент даних, використовують функцію **SizeOf**, яка має один параметр. Цей параметр може бути ім’ям типу чи змінної, константою або виразом. Наприклад:

```
WriteLn(SizeOf(Real));              {Друкується значення 6}
WriteLn(SizeOf(n))     {Друкується значення 4 n - LongInt}
```

1.7. Найпростіші введення і вивід

Найпростіше введення даних у ТР здійснюється з клавіатури, а вивід – на екран. Для цього служать оператори **Read, ReadLn** (введення) і **Write, WriteLn** (вивід). Один оператор введення може забезпечити введення значень відразу в декілька змінних, для чого ці змінні перелічуються в круглих дужках через кому:

Read(змінна1**,** змінна2**,...)**

або

ReadLn(змінна1**,** змінна2**,...).**

Аналогічний формат мають оператори виводу **Write** і **WriteLn**; однак у списку виводу можуть перелічуватися не тільки змінні, але і константи чи вирази.

Наприклад:

```
var
   n:Integer;
   r:Real;
begin
   ReadLn(n,r);
   WriteLn(n,' ',Sin(r))      {Виводяться: знач. змінної n,...}
end.          {... симв. константа "прогалина" і вираз Sin(r)}
```

Якщо в одному операторі введення перелічено декілька числових змінних, то значення, що вводяться, повинні розділятися або пробілом (можна декількома), або натисканням клавіш <Tab> чи <Enter>. Дані типу **Char** або взагалі не розділяються, або розділяються натисканням клавіші <Enter>. Відмінність операторів **Read** і **ReadLn** у наступному: якщо дані вводяться за допомогою **ReadLn**, то після набору останнього значення в списку введення потрібно обов'язково натиснути клавішу <Enter>; якщо ж дані вводяться оператором **Read**, то натискання <Enter> необхідно тільки для останнього оператора введення. Наприклад, якщо в наведеній вище програмі потрібно ввести значення 2 (*n*) і 1.2 (*r*), то рядок введення буде виглядати так: 2 1.2<Enter>. Аналогічно може виглядати рядок введення і для випадку **Read**(n); **Read**(r); у той же час для операторів **ReadLn**(n);**ReadLn**(r); необхідні два рядки введення:

2<Enter>

1.2<Enter>

Якщо оператори виводу **Write**, **WriteLn** виводять кілька елементів даних, то виведені значення ніяк не розділяються (у наведеній програмі для поділу спеціально виводиться прогалина). По закінченні виводу оператор **WriteLn** переводить курсор у початок наступного рядка, і наступне значення буде виводитися в новому рядку. Оператор **Write** по закінченні виводу курсор не переводить. Якщо при виводі дані не вміщуються в рядок, то вивід автоматично продовжується в наступному рядку; при досягненні останньої позиції останнього рядка екрана зображення переміщується на рядок вгору (це ж відбувається

при виведенні в останньому рядку за допомогою **WriteLn**). Оператор **WriteLn** без параметрів здійснює перехід до початку наступного рядка.

У TP під виведені значення на екрані відводиться мінімальна кількість позицій (своя для кожного типу даних), у які ще може бути здійснений вивід. В операторах виводу при кожному виведеному елементі можна вказувати форматний параметр, що задає кількість позицій, які відводяться на екрані під виведене значення. Він задається у вигляді цілого беззнакового числа через двокрапку, наприклад **WriteLn**(n:7,r:10). У цьому випадку виведене значення притискається до правого краю відведеного під нього поля, а зайві позиції зліва заповнюються прогалинами. Якщо відведене поле недостатнє для виводу, воно ігнорується. При виводі даних дійсного типу слідом за першим форматним параметром можна задати другий:

```
WriteLn(r:10:2);
```

Він указує, що значення повинне виводитися у вигляді числа з фіксованою крапкою (без порядку) із заданою цим параметром кількістю цифр після десяткової крапки. Так, оператор **WriteLn**(r:10:2) здійснює вивід значення r у 10 позиціях, з них 7 відводяться під знак і цілу частину, 1 – під десяткову крапку, 2 – під дробову частину. Виведене значення при цьому округляється.

1.8. Оголошення констант

1.8.1. Константи

Якщо в програмі використовується числовий матеріал, то у випадку її настроювання на нові числові значення доведеться здійснювати корекцію по всьому тексту. Більш розумно дати цим константам імена, що здійснюється визначенням констант. Визначення константи дозволяє наділити деяке ім’я значенням, котре може бути використане в будь-якому місці програми і не може бути змінене (насамперед, випадково). Воно здійснюється в спеціальному розділі, що має вигляд:

const
ім’я_константи1=значення1**;**
ім’я_константи2=значення2**;**

Константи можуть бути не тільки числовими, але і символьними, булевськими, рядковими. Наприклад:

```
const                                   {Оголошення констант:...}
   e=2.72;zero=0;       {...числових (дійсної і цілої),...}
   no=False;yes=True;                      {...булевських,...}
   background='-';                         {...символьної,...}
   language='Turbo Pascal 7.0';                {...рядкової}
```

Якщо потрібно модифікувати програму заміною деяких значень, досить внести відповідні зміни в розділ визначення констант. Наведена в прикладі 1.2 програма та її модифікація демонструють перевагу програм, які використовують константи.

```
{Приклад 1.2}
{Сума перших 30 натуральних чисел}
program sum_30;
var
    i,sum   : Integer;
begin
    sum:=0;
    for i:=1 to 30 do sum:=sum+i;
    WriteLn(sum, ' - це сума перших 30 чисел')
end.
```

Якщо обчислюється сума іншої кількості чисел, потрібно буде в трьох місцях внести зміни. Скористаємося визначенням констант для того, щоб програма стала гнучкою:

```
{Сума перших NumberToSum натуральних чисел}
const                                               {Константи:}
   NumberToSum=30;                                {...числова і...}
   message=' - це сума перших 30 чисел';               {...рядкова}
var
   i,sum     : Integer;
begin
   sum:=0;
   for i:=1 to NumberToSum do sum:=sum+i;
   WriteLn(sum,message)
end.
```

Тепер, якщо це буде потрібно, досить замінити дві константи на початку програми.

1.8.2. Типізовані константи

Додатково до звичайних констант є можливість використовувати типізовані константи, що є чимось проміжним між константами і змінними:

1. Типізовані константи описуються в розділі опису констант.

2. Як і звичайні константи, типізовані константи при описі одержують значення.

3. Аналогічно змінним типізовані константи мають тип, що задається описом.

4. Як і змінні, типізовані константи можуть змінювати своє значення.

Опис типізованої константи має вигляд

const

ім’я**:** тип=значення**;**

Значення типізованої константи може задаватися константою, константою посилального типу, ім’ям підпрограми, зображенням масиву, зображенням запису, зображенням множини, зображенням об’єкта.

Наприклад:

```
const
   zero: Integer=0;
   a   : Real=1.34e-7;
```

При описі типізованих констант повинна бути забезпечена сумісність за присвоюванням.

Принцип опису типізованих констант для більш складних типів буде розглянутий далі у розділах, присвячених відповідним типам даних.

1.9. Відрізки

Нехай існує тип даних, утворений сукупністю дискретних величин (наприклад, **Integer**). На основі цього типу можна побудувати новий тип, називаний відрізком. Наприклад, дані типу **Integer** мають значення від –32768 до 32767. Якщо нас цікавить більш вузький діапазон, то ми скористаємося відрізком (наприклад, від 1 до 47). Якщо задати відрізок, то компілятор буде виловлювати значення змінних-відрізків, що виходять за заданий діапазон, з діагностикою відповідної помилки. При цьому спрощується процес налагодження програми, але домогтися економії пам’яті не вдасться, бо елементи даних типу «відрізок» займають обсяг пам’яті, необхідний для розміщення кожного зі значень даного типу (наприклад, під змінну, яка має тип 200..300, буде виділено 2 байти пам’яті, бо її базовим типом буде тип **Integer**).

При описі відрізка вказують початок і кінець діапазону, розділяючи їх двома крапками:

```
var
    ім'я_відрізка: початок_діапазону..кінець_діапазону;
```

Наприклад:

```
var
   number :1..47;
   letter :'a'..'z';
```

Щоб компілятор виявляв вихід за діапазон значення змінної відрізкового типу, потрібно вмикати директиву компіляції **$R**: **{$R+}**. За замовчуванням вона вимкнена (**{$R-}**).

1.10. Розділ опису типів

Програміст може задати в програмі деякий нестандартний тип. При цьому виконуються два кроки: на першому в розділі опису типів, що починається зі службового слова **type**, задається зразок (шаблон) нового типу даних, на другому новий тип застосовується в оголошенні змінних:

```
const
  limit=47;
  len=5;
type                                    {Розділ опису типів}
  range=1..limit;
  StringLen=string[len];
var
  number: range;     {Використання раніше оголошених...}
  st    : StringLen;        {...типів range і StringLen}
```

Розділ опису типів є важливим розділом у програмі на ТР. По-перше, якщо тип заданий не ім'ям, а конструюється по ходу оголошення змінних (*var a:array[1..20] of Real*), то в програмі можлива несумісність типів у змінних, які, на перший погляд, мають той самий тип. Це може викликати помилки (у ТР прийнята іменна еквівалентність типів, про що буде говоритися в п.1.12.1). По-друге, зручніше спочатку оголосити ім'я для якогось типу, а потім це ім'я використовувати.

1.11. Перелічуваний тип

У ТР існує тип даних, значеннями якого є імена, які перелічені при оголошенні типу. Формат оголошення подібного роду типу (що зветься *перелічуваним* типом) такий:

type

ім’я_типу=**(**ім’я1**,** ім’я2**, ...);**

Можливими значеннями оголошеної в програмі змінної перелічуваного типу будуть імена, зазначені в дужках. Наприклад:

```
type
  Months=(January,February,March,April,Mai,June,July,
        August,September,October,November,December);
var
  Month:Months;
const
  Month1:Months=Mai;                {Типізована константа}
```

Значення перелічуваного типу упорядковані за зростанням відповідно до порядку їхнього переліку. Так, для типу Months справедливі нерівності January<February<...<December. До змінних перелічуваного типу можна застосовувати функції **Ord**, **Pred**, **Succ** і процедури **Inc** і **Dec**. Порядкові значення перелічуваного типу відраховуються від 0: застосування функції **Ord** до першого зі значень дає в результаті 0 (наприклад, **Ord**(January)=0, **Ord**(April)=3). Усього можна задати до 256 різних значень для одного перелічуваного типу.

```
{Приклад 1.3}
{Чи правда, що середньорічна зарплата була
 менше середньої зарплати за перше півріччя?}
{У програмі використаний оператор циклу for,
 який буде розглянутий у відповідному розділі}
type
   Months=(January,February,March,April,Mai,June,July,
         August,September,October,November,December);
var
   Month:Months;
   SumEmol1,SumEmol2:Real;      {Сумарний доход по півріччях}
   Emolument:Real;                          {Доход за місяць}
begin
   SumEmol1:=0;SumEmol2:=0;
   WriteLn('Введіть через <Enter> доход по місяцях');
   for Month:=January to December do begin {Цикл по місяцях}
      ReadLn(Emolument);
      if (January<=Month)and(Month<=June)        {Порівняння}
         then SumEmol1:=SumEmol1+Emolument
         else SumEmol2:=SumEmol2+Emolument
   end;
   WriteLn((SumEmol1+SumEmol2)/12<SumEmol1/6);      {Друк...}
end.                                        {...TRUE або FALSE}
```

Особливістю змінних перелічуваного типу є те, що їх значення не можна вводити з клавіатури і виводити на екран (але можна використовувати при роботі з типізованими файлами).

1.12. Еквівалентність і сумісність типів

1.12.1. Еквівалентність типів

У ТР суворо визначено, які типи описують ідентичні множини значень (еквівалентні). У мові прийнятий принцип іменної еквівалентності типів, який встановлює, що два типи T1 і T2 еквівалентні, якщо виконується одна з умов:

1) T1 і T2 — це одне й те ж ім'я типу;

2) тип T2 описаний з використанням типу T1 рівністю виду

```
type T2=T1;
```

чи послідовністю подібного виду рівностей:

```
type
   T1=Integer;
   T3=T1;
   T2=T3;
```

Наприклад, не є еквівалентними типи

```
type
   Tp1=array[1..10] of Real;
   Tp2=array[1..10] of Real;
```

незважаючи на їхню абсолютну ідентичність. У той же час при описі виду

```
type Tp3=Tp1;Tp4=Tp3;
```

пари типів Tp1 і Tp3, Tp4 і Tp3, Tp4 і Tp1 еквівалентні.

Якщо змінні описані спільно, то вони мають еквівалентні типи. Так, в описах

```
var
   v1,v2:array[1..10] of Real;
   v3,v4:Tp1;
```

у пар змінних v1 і v2, v3 і v4 типи еквівалентні, а в парах v1 і v3, v1 і v4, v2 і v3, v2 і v4 – типи не є еквівалентними.

1.12.2. Сумісність типів

Однією з вимог ТР є така: у виразах (у тому числі при порівнянні) повинні використовуватися операнди із сумісними типами. Типи

сумісні, якщо виконується хоча б одна з умов (не розглянуті до даного моменту типи будуть описані далі):

- обидва типи еквівалентні;
- обидва типи цілі;
- обидва типи дійсні;
- один з типів є відрізком, причому базовим для нього є інший тип:

```
type
   Tp1=LongInt;
   Tp2=-10..10;
```

- обидва типи є відрізками одного базового типу:

```
type
   Tp1='A'..'Z';
   Tp2='A'..'F';
```

- один тип рядковий, а другий – або рядковий, або символьний, або упакований символьний масив;
- обидва типи – упаковані символьні масиви з однаковим числом елементів;
- обидва типи множинні із сумісними базовими типами:

```
type
   Tp1=set of Byte;
   Tp2=set of 1..100;
```

- один тип є посилальним, а другий – або посилальний, або безтиповий покажчик;
- обидва типи – це процедурні типи з тим самим числом параметрів, причому типи параметрів повинні бути еквівалентними (відповідно до порядку, в якому вони зустрічаються), а для функціональних типів, крім того, повинні бути еквівалентними типи результатів.

Якщо у виразі типи сумісні, але різні, то тип результату визначається більш загальним з типів операндів.

Крім понять еквівалентності і сумісності типів, у ТР введене поняття сумісності за присвоюванням.

Оператор присвоювання коректний, якщо тип змінної в його лівій частині (Т1) є сполучимим за присвоюванням з типом виразу в правій частині (Т2). Для цього повинна виконуватися хоча б одна з умов:

- обидва типи еквівалентні, але жоден з них не є файловим типом чи складним типом, що використовує файловий тип;
- обидва типи – сумісні дискретні типи, і поточне значення типу T2 потрапляє в діапазон можливих значень типу T1;
- обидва типи дійсні, і поточне значення типу T2 потрапляє в діапазон можливих значень типу T1;
- тип лівої частини дійсний, а тип правої частини – цілий;
- тип T1 – рядковий тип, а T2 – або рядковий, або символьний, або упакований символьний масив;
- обидва типи – упаковані символьні масиви;
- обидва типи – сумісні множинні типи, причому множина з правої частини цілком входить у множину типу T1;
- обидва типи – сумісні посилальні типи;
- тип лівої частини – процедурний тип, а права частина – ім'я процедури чи функції з тим же числом параметрів, що й у типу лівої частини; типи відповідних параметрів (а також типи результату для функції) повинні бути еквівалентними;
- обидва типи – об'єктові типи, причому тип T2 є нащадком типу T1;
- обидва типи – посилальні типи на сумісні об'єктові типи.

Використання несумісних за присвоюванням типів спричиняє помилки.

1.13. Явне перетворення типів

У деяких випадках у TP відбувається автоматичний перехід від одного типу даних до іншого (від цілого до дійсного, від символьного до рядкового). Існує також ряд функцій, які здійснюють перетворення типів (**Ord**, **Chr**, **Trunc**, **Round**). Поряд з цим, у TP можливо явне перетворення типів (ретипизація даних). Для того щоб здійснити явне перетворення типу, необхідно використовувати ім'я типу аналогічно тому, як використовується ім'я функції. Як параметр у цьому випадку вказується ім'я змінної, що перетворюється :

```
var
   n:Byte;
   c:Char;
   m:LongInt;
```

```
begin
   c:=#0;
   n:=Byte(c);
   WriteLn(n,' ',Boolean(c));{Печать: 0 FALSE}
end.
```

Перетворювати можна будь-який тип до будь-якого іншого, однак треба виконувати вимогу: в операторі присвоювання змінна, що стоїть в лівій частині, повинна займати в пам'яті стільки ж або більше місця, скільки займає перетворене значення:

```
{Приклад 1.4}
{Обчислити значення функції
 sign(x)= 1 при x>0; 0 при x=0; -1 при x<0}
var
   x:Real;
   sign:Integer;
begin
   ReadLn(x);
   sign:=Byte(x>0)-Byte(x<0);
   WriteLn('x=',x,' sign=',sign)
end.
```

У програмі при x>0 вираз x>0 має значення **True**, а вираз x<0 – **False**, а це означає, що **Byte**(x>0)=1, **Byte**(x<0)=0 і sign=1; при x<0 вираз x>0 має значення **False**, а вираз x<0 – **True**, а це означає, що **Byte**(x>0)=0, **Byte**(x<0)=1 і sign=-1; при x=0 вирази x>0 і x<0 мають значення **False**, а це означає, що **Byte**(x>0)=0, **Byte**(x<0)=0 і sign=0;

В операторі присвоювання ім'я типу може зустрітися й у лівій частині. Наприклад,

```
var
   n:Byte;
   x:Real;
begin
   ReadLn(x);
   Boolean(n):=(x>0);
   {...}
end.
```

Тип даних, що використовується при цьому в лівій частині оператора присвоювання, повинен відповідати типу значення, яке стоїть у правій частині. У змінну в лівій частині записується значення не у форматі, що відповідає типу цієї змінної, а у форматі того типу, до якого йде перетворення.

1.14. Бітова арифметика

У ТР, крім звичайних дій над цілими і дійсними даними, введені додаткові операції над цілими типами (**Byte**, **ShortInt**, **Word**, **Integer**, **LongInt** і їхні діапазони) – бітова (порозрядна) арифметика.

1.14.1. Логічні операції над бітами

Якщо існує два цілих операнди A1 і A2, то над їх відповідними бітами можна виконувати раніше розглянуті логічні операції:

not – порозрядне заперечення;
and – логічне множення;
or – логічне додавання;
xor – виключаюче «чи».

Результат виконання кожної з цих операцій залежить від значень відповідних бітів операндів і визначається наведеною нижче таблицею.

A1	A2	**not** A1	A1 **and** A2	A1 **or** A2	A1 **xor** A2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Приклади

	6 **and** 4=4			6 **or** 4=4	
	00000110 (6)			00000110 (6)	
and	00000100 (4)		**or**	00000100 (4)	
	00000100 (4)			00000110 (6)	

Операція **and** практично завжди використовується тільки для досягнення однієї з двох цілей: перевірити наявність встановлених у 1 бітів чи здійснити обнуління деяких з бітів.

Подібна перевірка потрібна, коли число — це набір ознак з двома можливими значеннями (набір прапорів). Так, багато системних комірок пам’яті містять відомості про конфігурацію комп’ютера чи його стан. При цьому установка біта з конкретним номером у 1 трактується як вмикання якогось режиму, а в 0 – вимикання. Нехай змінна *A* має тип **Byte** і є байтом з вісьмома прапорами і потрібно

перевірити стан біта з номером 5 (справа наліво від 0 до 7). Одиниця в біті 5 – це п'ятий ступінь числа 2, тобто 32. Тому, якщо в п'ятому біті змінної *A* стоїть одиниця, то виконується умова (*A* **and** 32)=32, яку потрібно перевіряти в операторі **if**. Якщо необхідно перевіряти стан декількох одночасно встановлених у 1 бітів, то також потрібно обчислити відповідне число як суму ступенів числа 2, де показники ступеня дорівнюють номерам бітів, встановлених у 1. Наприклад, для бітів 5, 2 і 0 маємо 32+4+1=37. Якщо *A* має серед інших одиниці в бітах 5, 2 і 0, то виконується умова (*A* **and** 37)=37.

Приклади

Перевірка установки біта 5

```
  10110010  (178)
and
  00100000  ( 32)
  ---------------
  00100000  ( 32)  – Так
```

Перевірка установки бітів 0, 2, 5

```
  10110110  (182)
and
  00100101  ( 37)
  ---------------
  00100100  ( 36) – Не всі
```

Нехай потрібно записати нуль у який-небудь біт змінної *A* типу **Byte** (наприклад, у біт 3). Визначимо спочатку число, що містить одиниці у всіх бітах, крім 3-го. Максимальне число, яке можна записати в тип **Byte** — це 255. Щоб записати нуль у 3-й біт віднімемо від цього числа третій ступінь числа 2 (255-8). Якщо це число логічно помножити на *A*, то його одиниці ніяк не позначаться на стані змінної *A*, а 0 у третьому біті незалежно від початкового значення третього біта змінної *A* дасть у результаті 0. Отже, маємо *A*:=*A* **and** (255-8).

Аналогічно можна записати нулі у декілька бітів.

Приклади

Обнуління 3-го біта

```
  10111010  (188)
and
  11110111  (255-8)
  ---------------
  10110010  (172)
```

Обнуління декількох бітів (0, 2, 5)

```
  10110011
and
  11011010
  ---------
  10010010
```

Операцію **or** застосовують при установці в 1 окремих бітів двійкового подання цілих чисел. Так, щоб установити біт 4 змінної *A* у 1 без зміни інших бітів, досить записати *A*:=*A* **or** 16, де 16 – четвертий ступінь числа 2. Аналогічно встановлюють в 1 декілька бітів.

Операцію **xor** застосовують для зміни значення біта (чи декількох бітів) на протилежне (1 на 0 чи 0 на 1). Так, щоб переключити на протилежний стан біта 3 змінної *A*, треба записати *A*:=*A* **xor** 8, де 8 –

третій ступінь числа 2. Відзначимо, що дворазове застосування операції **xor** відновлює старе значення змінної: ((*A* **xor** *B*) **xor** *B*)=*A*.

Приклади

Установка декількох бітів (0 і 4)

```
   101101 0  (188)
or 000 0000 1  ( 17)
   ------------
   101 1101 1  (189)
```

Інверсія бітів 0 і 4

```
   101 1101 0  (188)
xor 000 1000 1  ( 17)
   ------------
   101 0001 1  (173)
```

1.14.2. Операції циклічного зміщення

У TP визначені ще дві операції над даними цілого типу, які мають той же пріоритетний рівень, що й операції **and**, *, /, **div** і **mod**. Це операції **shl** і **shr,** які зміщують послідовність бітів у двійковому поданні числа на задане число позицій вліво і вправо відповідно. При цьому біти, які виходять за розрядну сітку, губляться. При виконанні операції **shl** біти, що звільнилися праворуч, заповнюються нулями. При виконанні операції **shr** біти, які звільнилися ліворуч, заповнюються одиницями при зміщенні вправо від'ємних значень і нулями у випадку додатних значень.

Приклади

A **shl** 2 – циклічне зміщення вліво на два біти;

A **shr** 3 – циклічне зміщення вправо на три біти,

За допомогою операції **shl** можлива заміна операції множення цілих чисел на ступені двійки:

(J **shl** 1)=J*2, (J **shl** 2)=J*4, (J **shl** 3)=J*8.

```
{Приклад 1.5}
{Дане натуральне число n. Обчислити n-у ступінь числа 2}
var
   result:LongInt;
   n:byte;
begin
   result:=1;
   WriteLn('Введи показник ступеня (n<32): ');ReadLn(n);
   result:=result shl n;
   WriteLn(result);
end.
```

2. КЕРУЮЧІ КОНСТРУКЦІЇ

2.1. Найпростіший оператор, що перевіряє умову

Найчастіше в програмі необхідно виконувати деякі дії в тому випадку, коли істинна деяка умова. Щоб задати яке-небудь питання, у ТР застосовують оператор **if** (оператор умови):

if умова **then** оператор**;**

Якщо «умова» правдива (**True**), то виконується оператор, розташований після службового слова **then**, а потім виконується наступний оператор; якщо перевірка «умови» дає значення **False**, то оператор, що стоїть після **then**, пропускається. Як «умова» може стояти будь-який вираз, результатом обчислення якого є одне з булевських значень **True** чи **False**.

Відзначимо, що після **then** може стояти тільки один оператор. При необхідності виконання декількох операторів вони повинні бути укладені в операторні дужки **begin - end** (формується так називаний складений оператор).

Наведена конструкція зручна у випадку, коли деяка дія повинна бути виконана тільки при позитивній відповіді на питання і не виконуватися, якщо отримана негативна відповідь. Якщо ж при негативній відповіді необхідно виконати іншу дію, то керуючу конструкцію потрібно повторити з умовою, протилежною умові, яка перевіряється в першій конструкції:

if умова **then** оператор1**;**
if протилежна_умова **then** оператор2**;**

```
{Приклад 2.1}
{Обчислити y=-1, якщо x<0, y=1 у протилежному випадку}
program alternative;
var
  x,y:Real;
begin
  Write('Введіть число ');ReadLn(x);
  if x<0 then y:=-1;                {Перевірка умови}
  if x>=0 then y:=1;    {Перевірка протилежної умови}
  WriteLn('y=',y)
end.
```

Більш загальною конструкцією є конструкція **if-then-else**, ідея якої полягає в тому, щоб виконувати тільки один з пунктів — **then**

(якщо умова правдива) чи **else** (якщо неправдива), але ніколи не виконувати обох:

if умова **then** оператор1 **else** оператор2**;**

З використанням такої конструкції приведена вище програма прийме вид

```
{Обчислити y=-1, якщо x<0, y=1 у протилежному випадку}
program alternative1;
var
  x,y:Real;
begin
  Write('Введіть число ');ReadLn(x);
  if x<0 then y:=-1
         else y:=1;
  WriteLn('y=',y)
end.
```

Іноді деяку дію потрібно виконувати, тільки якщо умова, що перевіряється, помилкова. Тоді після **then** вказують порожній оператор:

if умова **then else** оператор**;**

У цьому випадку більш розумно представити умову, що перевіряється, у вигляді протилежної умови:

if протилежна_умова **then** оператор**;**

Наприклад, еквівалентними за одержуваним результатом є такі дві конструкції:

```
if x<x1 then else z:=x1;
```

і

```
if x>=x1 then z:=x1;
```

Слід зазначити, що і «оператор1», і «оператор2», які фігурують в операторі **if**, можуть бути будь-якими операторами, у тому числі операторами циклу (див. п.2.5) і операторами, що перевіряють умову. Тому, якщо необхідно вибрати одну з декількох взаємовиключних альтернатив, на допомогу приходить вкладений оператор **if**. Формат вкладеного **if** такий:

if умова1
  then оператор1
  else if умова2
    then оператор2
    else оператор3;

чи

```
if умова1
  then if умова2
          then оператор1
          else оператор2
  else оператор3;
```

Відповідність між **then** і **else** встановлюється в такий спосіб: кожному **then** відповідає найближче **else**, не задіяне при встановленні відповідності з іншим **then** (при відсутності між **then** та **else** слова **end**).

2.2. Оператор вибору

Оператор вибору, чи оператор **case**, можна трактувати як деяке питання, що має велике число відповідей (а не тільки дві, як це має місце в операторі **if-then-else**). Його формат такий:

```
case селектор of
    альтернатива1: оператор1;
    альтернатива2: оператор2;
          ...
    альтернативаN: операторN;
end;
```

Селектором може бути будь-який вираз порядкового типу (наприклад, **Integer**, **Char**, перелічуваний тип, але не **Real**). Поточне значення селектора визначає, який з операторів потрібно виконати. Елементи «альтернатива1», «альтернатива2»,..., «альтернативаN» є константними значеннями, які може приймати селектор. Їхній тип і тип селектора повинні бути сумісні за присвоюванням. Якщо селектор приймає значення «альтернатива1», то виконується «оператор1», а всі інші пункти пропускаються; якщо селектор приймає значення «альтернатива2», то виконується «оператор2» і т.д. Будь-який з цих операторів може бути або простим, або складеним, або порожнім (тільки крапка з комою).

```
{Приклад 2.2}
{Встановити, чи кратне дане ціле число трьом}
program mod3;
var
  number,remainder:Integer;       {Число і залишок від ділення}
begin
  Write('Введіть число ');ReadLn(number);
  remainder:=number mod 3;
```

```
  case remainder of
    0:WriteLn(number,' кратне 3');
    1:WriteLn(number,' кратне 3 із залишком 1');
    2:WriteLn(number,' кратне 3 із залишком 2');
  end {case}
end.
```

В альтернативі вибору можна вказати більш одного значення селектора (їх перелічують через кому), частиною альтернативи може бути відрізок (ряд послідовних значень), який задають вказівкою початку і кінця, розділених двома крапками (наприклад, 7..27).

```
{Приклад 2.3}
{Класифікувати малі латинські літери за правилом: літери b
 i d - клас 1; c,j,q - клас 2; a - клас 3;  h і всі
 літери від r до z - клас 4; інші літери - клас 5}
program classification;
var
  letter:'a'..'z';
  let_type:Integer;
begin
  WriteLn('Введіть літеру (a-z) для класифікації');
  ReadLn(letter);
  case letter of
    'b','d'              :let_type:=1;
    'c','j','q'          :let_type:=2;
    'a'                  :let_type:=3;
    'r'..'z','h'         :let_type:=4;
    'e'..'g','i','k'..'p':let_type:=5;
  end; {case}
  WriteLn('Літера ',letter,' у класі ',let_type)
end.
```

Обмеження:

1) селектор повинен мати який-небудь порядковий тип;

2) кожна альтернатива повинна бути константою, відрізком чи їх списком, але не змінною чи виразом.

Якщо серед альтернатив перелічені не всі можливі значення селектора, то при одержанні селектором такого значення оператор **case** фактично пропускається. ТР дозволяє згрупувати в пункті **else** усі значення, які не ввійшли в жодну з альтернатив.

```
{Приклад 2.4}
{Вводиться ціле додатне число – вік у літах.
 Вивести його разом з одним зі слів "рік", "роки", "років"}
var
   n:Word;
begin
   ReadLn(n);Write(n);
```

```
   if (n mod 100>10)and(n mod 100<20) then WriteLn('років')
   else case n mod 10 of
          1:WriteLn('рік');
          2,3,4:WriteLn(' роки');
          else WriteLn(' років');          {Те ж, що і 0,5...9}
        end;
end.
```

Перед **else** в операторі **case** крапка з комою може ставитися, а може не ставитися.

2.3. Оператор безумовного переходу

Призначення оператора переходу – порушувати природний порядок виконання операторів програми при виникненні яких-небудь особливих ситуацій. Він здійснює перехід до оператора, позначеного спеціальною міткою, яка відокремлюється від самого оператора двокрапкою. Як *мітка* може бути використане будь-яке ціле число без знака, що містить не більш чотирьох цифр, чи будь-яке ім'я. Щоб можна було використовувати мітку, вона повинна бути в обов'язковому порядку оголошена в розділі міток в описовій частині програми. Цей розділ починається службовим словом **label**, після якого через кому перелічуються мітки:

```
label 2, 1234,label_1,7777;
```

Щоб перейти до позначеного оператора, використовується оператор переходу, який має такий вигляд:

goto мітка**;**

Наприклад, передача керування може бути здійснена за допомогою оператора

```
goto label_1;
```

Однією з вимог структурного програмування є така: програма не повинна мати операторів переходу. Наявність у програмі великого числа операторів переходу свідчить про поганий стиль програмування.

2.4. Примусове припинення програми

Звичайно програма завершує свою роботу з досягненням її останнього оператора (тобто при виході на **end** із крапкою). Якщо виникає необхідність у припиненні виконання програми де-небудь

усередині її, то можна скористатися процедурою **Halt**, що викликається як окремий оператор. Цю процедуру можна викликати, задавши в круглих дужках параметр у вигляді цілого невід'ємного числа від 0 до 255. Це значення повертається в операційну систему у вигляді коду помилки (ERRORLEVEL) і може бути проаналізовано DOS у випадку запуску даної програми з командного файла. Відсутність параметра в процедурі **Halt** відповідає значенню параметра 0.

Другою процедурою, за допомогою якої можна припинити виконання програми, є процедура без параметрів **Exit** при її розміщенні в здійсненній частині програми (а не в тілі підпрограми). Частіше ця процедура застосовується для виходу з підпрограми без припинення виконання головної програми.

2.5. Цикли

Найчастіше в програмі доводиться організовувати багаторазове повторення одних і тих же операторів до виконання якої-небудь умови. Такі процеси називаються циклічними. Їх можна організувати з використанням операторів, що перевіряють умову, і оператора переходу. Однак у TP існують спеціальні оператори для організації циклів. Взагалі, існує два види циклів – цикл з передумовою і цикл з післяумовою. У першому випадку спочатку здійснюється перевірка деякої умови, і залежно від результату перевірки або виконується, або пропускається сукупність операторів, що утворюють тіло циклу. Якщо тіло циклу виконано, то процес повторюється, починаючи з перевірки умови. В другому випадку спочатку виконується тіло циклу, після чого здійснюється перевірка умови завершення циклу. Якщо умова не виконується, то процес повторюється. Особливим випадком циклу з передумовою є цикл з параметром, в якому тіло циклу виконується для послідовного ряду значень деякого параметра, який автоматично змінюється.

2.5.1. Цикл з передумовою

Цей вид циклу в TP задається конструкцією вигляду

while умова **do** тіло_циклу**;**

Спочатку обчислюється значення «умови» (це будь-який вираз, що дає або **True**, або **False**). Якщо воно істинне, виконується тіло

циклу, якщо неправдиве, – виконання циклу припиняється. Тіло циклу задається або одним оператором, або декількома операторами, укладеними в операторні дужки **begin - end**. У тілі циклу повинне змінюватися значення хоча б однієї змінної, що входить в «умову», інакше цикл буде нескінченним.

```
{Приклад 2.5}
{Знайти суму елементів ряду 1, -0.5, 0.25, -0.125,..., за
 модулем перевищуючих деяке додатне число}
program example_while;
var
   sum,epsilon,a:Real;
begin
   Write('Введіть граничне значення ');
   ReadLn(epsilon);
   a:=1; sum:=0;
   while Abs(a)>epsilon do begin
     sum:=sum+a;
     a:=-a/2
   end;
   WriteLn('Сума дорівнює ',sum)
end.
```

Оскільки в програмі використаний цикл з передумовою, то при введенні значення epsilon, більшого чи рівного одиниці, цикл виконуватися не буде, що дасть значення суми, яке дорівнює нулю.

```
{Приклад 2.6}
{Відомо, що на відрізку [a, b] знаходиться один
 корінь рівняння x+Sin(x)+f=0, де f - деяке значення.
 Знайти цей корінь методом ділення відрізка навпіл. Корінь -
 це будь-яка точка утримуючого його відрізка, довжина якого
 не перевершує заданого додатного числа}
var
  r,t,a,b,c,f,epsilon: Real;
begin
  Write('Введіть точність '); ReadLn(epsilon);
  Write('Введіть ліву границю відрізка '); ReadLn(a);
  Write('Введіть праву границю відрізка '); ReadLn(b);
  Write('Введіть вільний член f '); ReadLn(f);
  while Abs(b-a)>epsilon do
  begin
    c:=(b+a)/2;                              {Середина відрізка}
    if (a+Sin(a)+f)*(c+Sin(c)+f)<=0
       then b:=c       {Переміщення правої межі в середину...}
     {... відрізка, якщо корінь ліворуч від середини відрізка}
```

```
      else a:=c;                    {Інакше переміщуємо ліву межу}
  end;
  WriteLn('Значення кореня дорівнює ',c);
end.
```

2.5.2. Цикл з післяумовою

Цей вид циклу в ТР задається конструкцією вигляду

repeat тіло_циклу **until** умова**;**

Спочатку виконується тіло циклу (воно може складатися з декількох операторів, оскільки службові слова **repeat** і **until** обмежують тіло циклу з двох боків), після чого обчислюється значення «умови». Якщо воно дорівнює **False**, тіло циклу виконується повторно. Процес повторюється доти, поки значенням «умови» не стане **True**. Як і в циклі з передумовою, у тілі циклу з післяумовою повинно змінюватися значення хоча б однієї змінної, що входить в «умову».

```
{Приклад 2.7}
{З яких цифр складається ціле число}
var
   n:LongInt;
begin
   Write('Введіть ціле число ');ReadLn(n);
   if n<0 then n:=-n;                  {Знищення знака числа n}
   repeat
    WriteLn(n mod 10);     {Залишок від ділення на 10 - це ...}
                                  {... остання цифра числа n}
    n:=n div 10;             {Відкидаємо останню цифру числа n}
   until n=0;                             {Усі цифри розглянуті?}
   ReadLn
end.          {Програма працює при -2147483648<=n<=2147483647}
```

У даній програмі використаний цикл із післяумовою, оскільки хоча б один раз цикл повинен виконатися (будь-яке число складається хоча б з однієї цифри).

2.5.3. Цикл з параметром

У тих випадках, коли тіло циклу необхідно виконувати для послідовних значень деякої змінної якого-небудь дискретного типу, найчастіше використовується оператор **for**, або інакше – оператор циклу з параметром, що має один з таких двох форматів:

for параметр**:=**поч_знач **to** кінц_знач **do** тіло_циклу**;**

або

for параметр**:=**поч_знач **downto** кінц_знач **do** тіло_циклу**;**

«Параметр», чи інакше – індекс циклу, – це змінна будь-якого з дискретних типів (цілого, булевського, символьного, перелічуваного, відрізка); початкове і кінцеве значення – це вирази, сумісні за присвоюванням з параметром циклу; тіло циклу задається так само, як і в операторі **while**.

Цикл з параметром виконується в такий спосіб. Спочатку обчислюються початкове і кінцеве значення параметра циклу. Початкове значення присвоюється параметру циклу, а кінцеве – запам'ятовується, в результаті чого зміна усередині тіла циклу значення якої-небудь змінної, що входить у вираз для кінцевого значення, не позначається на кількості кроків циклу. Якщо запам'ятовано кінцеве значення більше або дорівнює поточному значенню параметра (у випадку **to** і менше або дорівнює при **downto**), то виконується тіло циклу, після чого параметр дістає наступне по черзі значення (у випадку **to** і попереднє при **downto**). Процес повторюється, і останній раз цикл виконується при значенні параметра циклу, яке вказане після **to** (**downto**). Так, якщо параметр циклу має який-небудь цілий тип, то при виконанні циклу він автоматично збільшується (при **to**) чи зменшується (при **downto**) на 1. Оскільки перевірка закінчення циклу здійснюється до першого виконання тіла циклу (цикл з передумовою), цикл з параметром може не виконуватися жодного разу.

```
 {Приклад 2.8}
{Вводяться цілі числа до першого числа, що менше
 двох. Скільки простих чисел було введено?}
var
   n,quantity,i:Integer;
   f: Boolean;                {True - число просте, False - ні}
begin
   quantity:=0;
   WriteLn('Вводіть цілі числа. Кінець введення – число <2')
   ReadLn(n);
   while n>2 do begin
      f:=True;
      for  i:=2  to n div 2 do
      if n mod i=0 then f:=False;       {Якщо n ділиться на...}
                       {...одне з чисел 2, 3,..., n div 2,...}
                                         {... то воно складене}
      if f then quantity:=quantity+1;          {Якщо True, ...}
      ReadLn(n);                              {...число просте}
```

```
   end;
   WriteLn('Було введено ',quantity,' простих чисел');
end.
```

У програмі показано, що можлива конструкція «цикл у циклі», чи інакше «вкладений цикл».

2.6. Оператори Break і Continue

Досить часто ціль виконання циклу досягається раніше, ніж він буде припинений за умовою виходу. Так, наприклад, в останній програмі внутрішній цикл буде виконуватися (n **div** 2-1) разів, хоча те, що число не є простим, може бути виявлено на перших кроках циклу. Щоб зменшити кількість кроків циклу, потрібно або скористатися оператором **goto**, або сформувати складну умову виконання (припинення) циклу; наприклад, в останній програмі цикл **for** повинен бути замінений циклом

```
while (i<=n div 2)and flag do
```

У TP 7.0 введена спеціальна процедура без параметрів **Break**, яка перериває утримуючий її цикл незалежно від виконання умови припинення (продовження) циклу. В останній програмі оператор **if**, який стоїть усередині оператора **for**, більш розумно записати так:

```
if n mod i=0 then begin
    flag:=True;Break
end;
```

що забезпечить припинення циклу **for** при першому ж діленні n на і без залишку.

Крім оператора **Break**, у TP 7.0 введена процедура без параметрів **Continue**, яка передає керування на кінець утримуючого її циклу, пропускаючи всі оператори, що стоять за нею.

Оператори **Break** і **Continue** виконуються в кожному з видів циклів (**repeat**, **while**, **for**) і дійсні тільки для самого внутрішнього з утримуючих їх циклів. Наприклад, якщо потрібно забезпечити примусовий вихід з подвійного циклу, оператор **Break** повинен бути розташований як у внутрішньому, так і в зовнішньому циклі. У деякому роді оператори **Break** і **Continue** – це сховані оператори **goto** з відомою точкою, в яку здійснюється перехід.

3. СКЛАДЕНІ ТИПИ

3.1. Масиви

3.1.1. Одновимірні масиви

У ТР існує механізм, що дуже просто розв'язує проблему позначення великого числа однорідних елементів. Не потрібно обмежуватися однією змінною, в яку по черзі заноситься безліч значень, чи оголошувати велику кількість різнойменних змінних. Для цього існує масив – змінна типу **array**, яка містить певне число елементів і дозволяє посилатися на перший, другий і т. д. елементи.

Для опису так називаного одновимірного масиву треба, крім його імені, вказати межі зміни значення індексу та тип елементів, який є спільним для всіх елементів масиву:

var ім'я**: array[**ниж_межа_інд **..** верх_межа_інд] **of** тип_елементів**;**

Наприклад:

```
var a,b: array[-10..100] of Integer;
    c  : array['a'..'z'] of Boolean;
```

Для звертання до елемента масиву вказують ім'я масиву й у квадратних дужках індекс (індекси), значення якого (яких) вказує на розташування елемента всередині масиву. Найчастіше використовують цілочислові індекси, але в загальному випадку індексами масиву можуть бути дані будь-якого порядкового типу, тобто такого, в якому, по-перше, визначена дискретна послідовність значень і, по-друге, усі ці значення можна перелічити один по одному. Індексувати можна як константами і змінними, так і виразами, результат обчислення яких дає значення порядкового типу.

Якщо індекс масиву може здобувати всі можливі значення деякого порядкового типу, то при описі масиву можливе завдання імені типу замість меж зміни індексу (при цьому межами індексу будуть перше й останнє значення в описі типу індексу). Межі зміни індексів можуть задаватися з використанням раніше оголошених констант. Рекомендується попередньо оголошувати тип масиву в розділі опису типів:

```
const
      num=150;
type
      DaysOfWeek=(Monday,Tuesday,Wednesday,Thursday,
                  Friday,Saturday,Sunday);      {Тип переліку}
      Years=1993..2003;                             {Відрізок}
```

```
      TTax=array[Years] of Real;                    {Тип масиву}
var                                  {Можливі методи опису масиву}
      gain:array[DaysOfWeek] of Real;
      TaxOfYear:TTax;
      production:array[1..num] of string[10];
```

Функція **SizeOf**, застосована до імені масиву чи імені масивного типу, повертає кількість байт, яка відводиться під масив. Так, оператори **Write(SizeOf**(TaxOfYear)) і **Write(SizeOf**(TTax)) виведуть значення 66.

При описі масиву і звертанні до його елементів квадратні дужки можуть бути замінені парою складених символів **(.** та **.)** .

```
{Приклад 3.1}
{Які різні цифри входять у ціле число N?}
var
   n:LongInt;
   numerals:array[1..10] of Byte;                  {Різні цифри}
   i,
   QuantityOfNumerals,                            {Кількість цифр}
   numeral:Byte;                                 {Виділювана цифра}
   flag:Boolean;
begin
   Write('Введіть ціле число ');ReadLn(n);
   QuantityOfNumerals:=0;
   repeat
     numeral:=Abs(n mod 10);
     flag:=False;    {Вважаємо, що цифра раніше не виділялася}
     for i:= 1 to QuantityOfNumerals do
       if numeral=numerals[i] then  begin
          flag:=True;Break;
       end;
     if flag=False then begin
       QuantityOfNumerals:=QuantityOfNumerals+1;
       numerals[QuantityOfNumerals]:=numeral;
     end;
     n:=n div 10
   until n=0;
   for i:=1 to QuantityOfNumerals do WriteLn(numerals[i]);
end.
```

Щоб описати типізовану константу-одновимірний масив, потрібно в круглих дужках вказати через кому значення всіх її елементів. При цьому тип масиву краще оголошувати в розділі опису типів:

```
   type
          t_flags=array[1..5] of Boolean;
   const
          flags:t_flags=(True,False,True,True,False);
```

```
{Приклад 3.2}
{В які дні доход був вище середнього доходу за тиждень?}
type
       DaysOfWeek=(Monday,Tuesday,Wednesday,Thursday,
                   Friday,Saturday,Sunday);
const  ukr_days:array[DaysOfWeek] of string[11]=
          ('Понеділок','Вівторок','Середа','Четвер',
           'П''ятниця','Субота','Неділя');
var    gain:array[DaysOfWeek] of Real;   {Доход по днях тижня}
       sum:Real;
       i:DaysOfWeek;
begin
  sum:=0;
  for i:=Monday to Sunday do begin
    WriteLn('Введіть через <Enter> доход за ',ukr_days[i]);
    ReadLn(gain[i]);sum:=sum+gain[i]
  end;
  WriteLn('Вище за середній доход був в');
  for i:=Monday to Sunday do
    if gain[i]>sum/7 then WriteLn(ukr_days[i])
end.
```

У даній програмі оголошена типізована константа-масив (змінна ukr_days). Діапазон зміни індексів у масивах ukr_days і gain задається перелічуваним типом DaysOfWeek. Природно, індексування елементів цих масивів також здійснюється значеннями типу DaysOfWeek.

```
{Приклад 3.3}
{Упорядкувати масив чисел за неубуванням методом
 "бульбашки". Кількість чисел - не більше 50}
var
   a:array [1..50] of Real;
   i,j,n:Integer;
   b:Real;
begin
   WriteLn('Введіть кількість елементів'); ReadLn(n);
   WriteLn('Введіть через <Enter> ',n,' чисел');
   for i:=1 to n do ReadLn(a[i]);
   for i:=1 to n-1 do
      for j:=n downto i+1 do
        if a[j]<a[j-1] then begin
           b:=a[j];
           a[j]:=a[j-1];
           a[j-1]:=b
      end;
   for i:=1 to n do Write(a[i],' ');
   ReadLn                          {Чекання натискання <Enter>}
end.
```

Метод («бульбашки») полягає в наступному. Береться останній елемент («бульбашка») і, якщо він менше («легше») попереднього, елементи переставляються, після чого здійснюється перехід до попереднього елемента, незалежно від того, була чи не була перестановка. Процес повторюється до порівняння другого і першого елементів. У результаті найменший («найлегший») елемент займе першу позицію в масиві. На другому кроці описані дії повторюються до моменту порівняння третього і другого елементів (перший елемент розглядати не потрібно, бо він вже найменший). Таким чином, з кожним кроком кількість розглянутих елементів зменшується на 1, і всього потрібно виконати N-1 кроків (поки не будуть розглянуті тільки останні два з N елементів).

3.1.2. Багатовимірні масиви

Ми розглядали одновимірний масив. Але дуже часто структуру даних зручно подати у вигляді прямокутної таблиці. Щоб звернутися до елемента такої структури, потрібно вказати дві координати – по горизонталі і по вертикалі. Для подання таких структур у ТР використовують двовимірні масиви. Відмінність в описі двовимірного масиву від одновимірного складається тільки в тому, що необхідно вказати верхню і нижню межі другого індексу. При цьому перший індекс служить для звертання до рядків, а другий – до стовпців масиву. Крім звичайного способу опису двовимірного масиву, можливий також розгляд його при описанні як масиву масивів Наприклад, такі два опису ідентичні:

```
var
   eval:array[1..100,1..4] of Byte;
var
   eval:array[1..100] of array[1..4] of Byte;
```

При використанні другого варіанта опису краще спочатку визначити деякий тип одновимірного масиву (рядок двовимірного масиву), що потім використовується при описі двовимірного масиву:

```
type
   exam=array[1..4] of Byte;
var
   eval:array[1..100] of exam;
```

Для звертання до елемента двовимірного масиву необхідно вказати ім’я масиву й у квадратних дужках через кому значення двох індексів: перший вказує номер рядка, а другий – номер стовпця, на

перетинанні яких стоїть елемент (eval[i,4]:=5). Можна застосовувати змішане індексування (наприклад, перший індекс має цілий тип, а другий – символьний). Індекси можуть розміщуватися кожний у своїх квадратних дужках без поділу комою (eval[1][4]:=5).

```
{Приклад 3.4}
{Даний квадратний числовий масив розміру не більше 100x100,
 усі елементи якого різні. Чи правда, що мінімальний з
 елементів, які стоять над головною діагоналлю і на ній,
 розташований правіше і нижче максимального елемента масиву?
 Головна діагональ прямує з лівого верхнього кута в правий
 нижній}
var
     a:array[1..100,1..100] of Real;
     i,j,n: Integer;
     StrMax,ColMax,{Номери рядків і стовпців максимального...}
     StrMin,ColMin:Integer;      {...і мінімального елементів}
begin
   Write('Введіть розмірність масиву (до 100) ');
   ReadLn(n);
    WriteLn('Введіть рядками масив');
   for i:=1 to n do
    for j:=1 to n do
       Read(a[i,j]);
   StrMax:=1;ColMax:=1;
   StrMin:=1;ColMin:=1;
   for i:=1 to n do
     for j:=1 to n do begin
       if a[i,j]>a[StrMax,ColMax] then begin
          StrMax:=i;ColMax:=j
       end;
       if a[i,j]<a[StrMin,ColMin] then begin
          StrMin:=i;ColMin:=j
       end;
     end;
   if (ColMin>ColMax)and(StrMin>StrMax) then WriteLn('Правда')
                                        else WriteLn('Неправда')
end.
```

Масиви можуть мати розмірність більше двох. При цьому в пам'яті вони розташовуються так, що останній індекс змінюється швидше всіх попередніх, а перший – повільніше усіх.

Якщо описується двовимірний масив як типізована константа, то при завданні значень його елементів він розглядається як масив масивів. При цьому в спільних круглих дужках через кому перелічуються взяті в круглі дужки значення елементів рядків (кожен рядок у своїх дужках):

```
type
  t_2arr=array[1..4,1..3] of Real;
const
  coord_points:t_2arr=((1.2, -0.5, 3.4),
    (0.0, 0.0, 0.0),(5.0, 4.1, 0.0),(3.7, -1.2, -1.0));
```

Аналогічно описуються типізовані константи – багатовимірні масиви (зовнішні дужки відповідають першому індексу, самі внутрішні – останньому):

```
const
  v_3arr:array[1..2,'A'..'C',-1..2] of Real=
   (((1.1,1.2,1.3,1.4),(2.1,2.2,2.3,2.4),(3.1,3.2,3.3,3.4)),
   ((4.1,4.2,4.3,4.4),(5.1,5.2,5.3,5.4),(6.1,6.2,6.3,6.4)));
```

3.1.3. Автоматичний контроль значень індексів

Однією з послуг, що надає компілятор TP, є автоматичний контроль значень індексів під час виконання програми. Для цього досить активізувати директиву компіляції **$R**, що дає вказівку компілятору здійснювати вбудований контроль граничних значень змінних. Найчастіше ця директива вимкнена, що зменшує обсяг програми і час обчислень. При активізації цієї директиви (**{$R+}**) усяка спроба використовувати індексне значення поза оголошеними межами буде зареєстрована як помилка часу виконання програми, бо в цьому випадку забезпечується вмикання контролю виходу значень порядкових змінних за діапазон відповідного типу. Наприклад, щоб у якому-небудь фрагменті програми здійснювався контроль діапазону, потрібно обмежити цей фрагмент директивою **$R**:

{$R+}
Фрагмент програми
{$R–}

3.2. Робота з рядками на TP 7.0

3.2.1. Поняття рядкових змінних. Присвоювання і введення/вивід

У TP є тип **Char**, значеннями якого є символи (літери, цифри, різні знаки). Найчастіше ж потрібно працювати не з окремими символами, а з рядками символів (наприклад, прізвище). TP дозволяє оголосити змінні, у яких можуть зберігатися рядки символів. Оголошення буде мати вигляд

var ім'я **:string;**

У цьому випадку для збереження значення змінної виділяється обсяг пам’яті в 255 байт, у кожному з байтів якого може зберігатися тільки один символ (тобто кожен елемент рядка фактично має тип **Char**). На самому початку області виділяється додатковий байт для збереження фактичної поточної довжини рядкового значення, записаного в рядок. Таким чином, під змінну типу **string** насправді виділяється 256 байт.

Якщо при оголошенні рядкової змінної після службового слова **string** у квадратних дужках вказати ціле додатне значення (наприклад, *var s:string[20]*), то це означає, що під відповідну рядкову змінну виділяється не 255 байт, а стільки, скільки зазначено в описі, і максимальна довжина рядка не може перевищити зазначеної. Замість квадратних дужок можна використовувати пари символів **(.** та **.)** так само, як і при описі масивів.

Для надання рядковій змінній значення можна скористатися оператором присвоювання чи читання з клавіатури або з файла. Наприклад:

```
var
   st :string;
   st1:string[20];
   st2:string[12];
   st3:string[4];
begin
   st:='Блез Паскаль';
   ReadLn(st1);
      ...
```

Оголошення типізованої константи для типу **string** здійснюється звичайним способом:

```
const
   s:string='Turbo Pascal 7.0';
```

Механізм рядкових присвоєнь має деякі особливості:

1. Якщо довжини одержувача і джерела однакові, проблем немає:

```
st2:='Блез Паскаль';
```

2. Якщо одержувач є довшим за джерело (st1:=st2), то фактична довжина одержувача встановлюється рівною довжині рядка-джерела (таким чином, фактична довжина рядка st1 виявиться рівною 12, а значенням буде ’Блез Паскаль’).

3. Якщо одержувач коротше джерела (st3:=st2), то поточна довжина рядка-одержувача виявиться рівною максимальній в

оголошенні, а значенням виявиться усіченим справа на потрібне число символів значення рядка-джерела (в даному випадку ’Блез’).

При виводі рядкових значень можна вказувати поле виводу, яке відводиться під це значення. Якщо довжина поля не зазначена, то вона приймається рівною поточній довжині рядка. Якщо довжина поля більша поточної довжини рядка, то відбувається вирівнювання по правому краю (вільні позиції зліва заповнюються прогалинами); якщо довжина поля недостатня для виводу, то поле виводу ігнорується. Наприклад, оператори

```
WriteLn(st2);WriteLn(st2:14);WriteLn(st2:4);
```

забезпечують відповідно такі варіанти виведення на екран:

```
Блез Паскаль
  Блез Паскаль
Блез Паскаль
```

Як це відзначалося раніше, у ТР є поняття *порожнього рядка* (рядок, який не має елементів). Порожній рядок позначається двома апострофами, що стоять поруч (наприклад, *st:=’’* чи *if st=’’ then*).

3.2.2. Порівняння рядків і їхня конкатенація

За правилами ТР за допомогою операцій =, <, <=, >, >=, <> дозволяється порівнювати рядкові змінні, константи і вирази з будь-якими максимальними і поточними довжинами. При цьому для встановлення факту рівності необхідно, щоб порівнювані об’єкти мали однакові фактичні довжини і точно співпадаючі значення символів. Коли в програмі порівнюються два рядки, в дійсності відбувається серія попарних порівнянь їхніх символів зліва направо до першої розбіжності чи вичерпання одного з рядків. Меншим буде той рядок, у якого менше код першого незбіжного символу (поза залежністю від максимальних і поточних довжин порівнюваних рядків). Якщо один рядок збігається з початком іншого, то більшим буде більш довгий рядок (наприклад, правдиві співвідношення ’Іван’<’Іваненко’ і ’джерело’>’джерельце’).

Два рядки можуть бути зчеплені один з одним за допомогою операції конкатенації (зчеплення), яка позначається символом «+».

Так, виконання операторів

```
st2:='Pascal'; st:='Turbo'+st2+' 7.0';
```

дасть результат ’Turbo Pascal 7.0’, записаний у st.

Операція конкатенації має більш високий пріоритет у порівнянні з операціями відносини <, <=, >, >=, =, <>.

3.2.3. Робота з окремими елементами рядка

При роботі з рядками можна звертатися до окремих їхніх елементів (символів) аналогічно тому, як це робиться при обробці елементів масиву. Окремі елементи змінної типу **string** мають усі властивості змінної типу **Char**.

Можна здійснювати корекцію будь-якого елемента рядкової змінної, для чого у відповідному операторі досить вказати ім'я змінної типу **string**, слідом за яким у квадратних дужках задається номер її елемента (наприклад, *st[10]:='f'*). Пари символів **(.** та **.)** і тут замінюють квадратні дужки. Елементи рядка нумеруються від 1, але в кожній рядковій змінній є елемент з номером 0, у якому у вигляді символу зберігається поточна довжина рядка. Щоб довідатися про поточну довжину, досить застосувати функцію **Ord** до нульового елемента рядка, наприклад *WriteLn(Ord(st[0]))*. Нульовий елемент рядкової змінної можна корегувати; при цьому буде змінюватися поточна довжина рядка (наприклад, оператор *st[0]:=#65* встановлює поточну довжину рівною 65).

```
{Приклад 3.5}
{Даний рядок. Якщо в ньому останні символи однакові, залишити
 тільки один з них, усунувши дублювання символів}
var
  st:string;
begin
  WriteLn('Введи рядок');
  ReadLn(st);
  while (st[0]>#1)and(st[Ord(st[0])]=st[Ord(Pred(st[0]))]) do
     Dec(st[0]);   {Те ж, що і st[0]:=Chr(Ord(st[0])-1); ...}
                                    {...чи st[0]:=Pred(st[0])}
  WriteLn(st);
end.
```

У цій програмі нерівність st[0]>#1 (порівняння символів) еквівалентна нерівності Length(st)>1. Номер останнього символу рядка Ord(st[0]), а для доступу до передостаннього символу використана функція **Pred**: номер передостаннього символу дорівнює Ord(Pred(st[0])). Робота програми заснована на послідовному зменшенні довжини рядки за рахунок зміни символу st[0].

3.2.4. Процедури і функції для обробки рядків

Замість операції конкатенації в рядкових виразах можливе використання функції **Concat**, в якій параметрами може виступати будь-яка кількість рядків:

ім'я_рядка**:=Concat(**рядок1**,** рядок2**,** рядок3**,...);**

Її значення, що повертається, – рядок довжиною не більш 255 символів, який утвориться приєднанням значень параметрів: другого до кінця першого, третього до результату попереднього приєднання і т. д. Наприклад:

```
st:=Concat(st1,st2,st3);
```

Щоб довідатися про поточну довжину рядка, можна скористатися функцією **Length**, яка повертає кількість символів, що містяться в даний момент у рядку, вказаному як параметр:

змінна_цілого_типу**:=Length(**рядок**);**

Для пошуку деякого сполучення символів у рядку в TP введена функція **Pos**, яка має такий формат:

p**:=Pos(**шуканий_рядок**,** оброблюваний_рядок**);**

Обидва параметри можуть бути будь-якими рядковими виразами; значення, що повертається, має тип **Integer**.

Якщо, наприклад, необхідно знайти рядок st1 у рядку st, то при звертанні p:=Pos(st1,st) у змінну цілого типу p буде записане число, що вказує ту позицію рядка st, у якій була виявлена перша поява рядка st1. Якщо st1 не виявляється в st, то результатом буде 0.

Функція **Copy** використовується в TP для копіювання частини рядка. Її формат:

рядок1**:=Copy(**рядок**,** початкова_позиція**,** кількість_символів**);**

Функція повертає рядок, утворений зазначеним числом символів (третій параметр), розташованих у зазначеному рядковому виразі (перший параметр) підряд, починаючи з позиції, номер якої задається другим параметром. Наприклад:

```
st:=Copy('утворений',2,3);        {st='твор'}
```

Якщо значення другого параметра задає позицію за межами поточної довжини початкового рядка, то повертається порожній рядок. Значення початкової позиції поза відрізком 1..255 дає помилку. Якщо третій параметр дає вихід за поточну довжину рядка, то копіюються тільки реально існуючі символи:

```
st:=Copy('Turbo Pascal',7,Length('Turbo Pascal'));{st='Pascal'}
```

```
{Приклад 3.6}
{Ввести рядковий масив з 10 елементів вигляду
 "Прізвище Ім'я". Перетворити всі рядки до вигляду
 "Ім'я Прізвище", після чого вивести їх на екран}
var
  s:array[1..10] of string;
  k,i:Byte;
begin
  for i:=1 to 10 do begin
    WriteLn('Введи: Прізвище Ім''я');
    ReadLn(s[i])
  end;
  for i:=1 to 10 do begin
    k:=Pos(' ',s[i]);                  {Пошук позиції прогалини}
    s[i]:=Concat(Copy(s[i],k+1,Length(s[i])),
            ' ',Copy(s[i],1,k-1))        {Можна   s[i]:=      }
          { Copy(s[i],k+1,Length(s[i]))+' '+Copy(s[i],1,k-1)}
  end;
  for i:=1 to 10 do
    WriteLn(s[i])
end.
```

Для видалення частини рядка використовується процедура **Delete**, яка має три параметри:

Delete(рядок, поч_позиція, кількість_ символів_що_видаляються**);**

Перший параметр – це рядкова змінна, два інших – будь-які цілочислові вирази.

Наприклад, два оператори *st:='метеор';Delete(st,4,2)* записують у st значення 'метр'.

При застосуванні процедури **Delete** здійснюється видалення заданого числа символів, починаючи з зазначеної позиції, зі зсувом на позиції, що звільнилися, символів, розташованих правіше вилучених, і зміною поточної довжини рядка. Якщо другий параметр більше поточної довжини, видалення не відбувається, якщо ж його значення знаходиться поза відрізком 1..255, виникає помилка.

Процедура **Insert** здійснює вставку рядка в деякий інший рядок, починаючи з зазначеної позиції, попередньо відсунувши вправо усе, що заважає вставці. Формат процедури:

Insert(рядок_що_вставляється**,** рядок_що_приймає**,** поз_вставки**);**

Якщо максимальна довжина рядка, в який провадиться вставка, менше результуючої довжини, то відбувається усікання результату справа. Якщо позиція вставки більше поточної довжини рядка, то перший параметр приєднується до кінця другого. При значенні третього параметра поза відрізком 1..255 фіксується помилка.

Наприклад, два оператори `st:='симу';Insert('вол',st,4);` забезпечують запис у st значення `'символу'`.

```
{Приклад 3.7}
{Замінити в рядку всі сполучення символів 'Turbo Pascal'
 на сполучення символів 'TP 7.0'}
var
  s:string;
  k:Byte;
begin
  ReadLn(s);
  k:=Pos('Turbo Pascal',s);            {k – позиція підрядка}
  while k<>0 do begin
    Delete(s,k,Length('Turbo Pascal')); {Видаляємо підрядок}
    Insert('TP 7.0',s,k);              {Вставка в k-у позицію}
    k:=Pos('Turbo Pascal',s);               {Повторюємо пошук}
  end;
  WriteLn(s)
end.
```

У деяких задачах виникає необхідність у перетворенні числових даних, записаних у рядковому вигляді, до якого-небудь числового формату. Для цих цілей у TP введена процедура **Val**, формат звертання до якої такий:

Val(рядок**,** числова_змінна**,** цілочислова_змінна**);**

Ця процедура присвоює другому аргументу числове значення, що зберігається в рядковому виразі, записаному як перший аргумент. Якщо під час перетворення не виявлена помилка, то в третій параметр записується 0; якщо помилка виявлена, то в третій параметр записується номер позиції першого помилкового символу рядка, а значення другого параметра буде дорівнювати нулю. Другий параметр може мати будь-який числовий тип, і саме до цього типу буде здійснюватися перетворення. Нормальне перетворення до цілого типу здійснюється тільки при приналежності числа, що утворюється, діапазону даних типу **LongInt**; однак контроль над тим, який цілий тип **(ShortInt, Byte, Integer, Word, LongInt**) має другий параметр, не провадиться. В перетвореному рядку можуть бути лідируючі прогалини; прогалини ж наприкінці рядка призводять до помилкового перетворення.

Наприклад, нехай у програмі є описи:

```
var
   r:Real;
   m,p:Integer;
   n:LongInt;
```

Нижче наведені приклади використання процедури **Val**:

```
Val('-12'.m,p);              {m=-12; p=0}
Val('100023',n,p);         {m=100023; p=0}
Val('-12e+004',r.p);   {r=-1.2e-05; p=0}
Val('-12e+004',n,p);           {n=0; p=4}
Val('   5623',n,p);         {n=5623; p=0}
Val('5623   ',n,p)             {n=0; p=5}
```

```
{Приклад 3.8}
{Рядок складається зі слів (сполучень символів, відділених
 один від одного однією чи декількома прогалинами). Вибрати ті
 зі слів, що є записом чисел типу LongInt, а ті, що залишилися,
 розсортувати на слова, що становлять собою числа типу Real,
 і слова, що не є числами в TP}
var
   s,s_L,s_R,s_Str,s1:string;{s_L - слова для типу LongInt...}
             {...s_R - слова для типу Real,s_Str - інші слова}
   k,p:Integer;
   r:Real;
   l:LongInt;
begin
   WriteLn('Введи рядок');ReadLn(s);
   while s[1]=' ' do
      Delete(s,1,1);      {Видалення прогалин: початкових,...}
   while s[Length(s)]=' ' do
      Delete(s,Length(s),1);                {...кінцевих і...}
   while Pos('  ',s)<>0 do
      Delete(s,Pos('  ',s),1);          {... тих, що двояться}
   s_L:='';s_R:='';s_Str:='';
   while s<>'' do begin
     k:=Pos(' ',s);
     if k=0 then k:=Length(s)+1;    {Обробка останнього слова}
     s1:=Copy(s,1,k-1);                        {Виділення слова}
     Val(s1,l,p);               {Перетворення до типу LongInt}
     if p=0 then s_L:=s_L+s1+' '                {Перетворилося}
     else begin                                          {Ні}
       Val(s1,r,p);                {Перетворення до типу Real}
       if p=0 then s_R:=s_R+s1+' '              {Перетворилося}
              else s_Str:=s_Str+s1+' ';                  {Ні}
     end;
     Delete(s,1,k);       {Видалення слова разом з прогалиною}
   end;      {В s_L, s_R та s_Str останній символ - прогалина}
   Dec(s_L[0]);Dec(s_R[0]);Dec(s_Str[0]);     {Видаляємо його}
   WriteLn('Цілі числа: ',s_L);
   WriteLn('Дійсні числа: ',s_R);
   WriteLn('Слова: ',s_Str);
end.
```

Зворотною стосовно процедури **Val** є процедура **Str**, формат якої такий:

Str(арифметичний_вираз**,** рядок**);**

Процедура перетворить значення свого першого аргументу в рядкову змінну, задану як другий параметр. Знак «+» у рядок не заноситься. Після арифметичного виразу може бути зазначеним формат, аналогічний формату виводу в операторах виводу.

Наприклад:

```
var st:string[7];
...
st:=Str(15-2*3,st);   {st='9', поточна довжина 1}
```

```
{Приклад 3.9}
{Дано натуральне число n. Одержати символьне подання
 цього числа у вигляді рядка цифр і прогалин, що
 відокремлюють групи по три цифри, починаючи справа}
var
    n:1..2147483647;           {Максимальний діапазон для...}
    n_str:string[13];                  {...натуральних чисел}
    i:Integer;
begin
    WriteLn('Введи число від 1 до 2147483647');
    ReadLn(n);
    Str(n,n_str);
    i:=Length(n_str)-2;
    while i>1 do begin
      Insert(' ',n_str,i);
      i:=i-3
    end;
    WriteLn(n_str);
end.
```

При роботі із символами в рядках, а також з окремими символами, досить часто виникає необхідність у переході від малих латинських літер до прописних. Для цього передбачена функція **UpCase**, яка має такий формат:

символьна_змінна**:=UpCase(**символ**);**

Функція не здійснює зворотне перетворення так само, як не перетворює ніякі інші символи, у тому числі українські літери.

```
{Приклад 3.10}
{Даний рядок, який складається з латинських літер (великих і
 малих) і прогалин. Чи правда, що він однаково читається
 справа наліво і зліва направо? }
```

```
var
    s:string;
    i:Integer;
    f:Boolean;
begin
    WriteLn('Введи рядок');
    ReadLn(s);
    i:=Length(s);
    while i>=1 do begin
      if s[i]=' '
         then Delete(s,i,1)                {Видалення прогалин}
         else s[i]:=UpCase(s[i]);    {Перехід до великих літер}
      i:=i-1
    end;
    f:=True;
    i:=1;
    while  f and(i<=Length(s) div 2) do begin
      if s[i]<>s[Length(s)-i+1] then f:=False;
      i:=i+1
    end;
    WriteLn(f)
end.
```

3.2.5. Рядки і масиви символів

У TP змінна типу **array of Char** може розглядатися як рядок постійної довжини. Змінні такого типу можуть вільно використовуватися в будь-яких рядкових виразах. При цьому компілятор автоматично перетворить такий масив у рядок, довжина якого дорівнює кількості елементів масиву. Масиви типу **array of Char** можна порівнювати один з одним і взагалі оперувати ними так само, як і змінними типу **string** (але з деякими обмеженнями). Можна в операторі присвоювання в лівій частині вказувати ім'я масиву типу **array of Char**, а в правій – рядкову константу з довжиною, яка дорівнює кількості елементів у масиві. У той же час у TP 7.0 масивам типу **array of Char** дозволено присвоювати рядкові константи, довжина яких менше кількості елементів масиву; в елементи, що залишилися при цьому незаповненими, заноситься символ #0, але для цього потрібно включити розширений синтаксис (директива **{$X+}**). Однак не можна змінній типу **array of Char** присвоїти значення рядкової змінної чи рядкового виразу (крім виразів над рядковими константами з результуючою довжиною, що дорівнює розмірності масиву). Масиви типу **array of Char** можуть використовуватися в процедурі **Val** і функціях **Concat**, **Copy** і **Length**.

3.2.6. Тип PChar

У TP 7.0 введений тип **PChar**, що описує так називані довгі (чи інакше ASCIIZ) рядки. У змінних типу **string** довжина рядка записується в нульовому байті, а оскільки в 1 байт не можна записати значення, яке перевершує 255, то і довжина змінної типу **string** не може перевершувати 255 символів. Змінна типу **PChar** – це рядок, що може мати довільну довжину (але не більш 65534 символів). Довжина таких рядків не вказується явно: рядок починається з першого символу й обмежується замикаючим символом #0. Оскільки тип **PChar** визначається як покажчик на символ, ASCIIZ-рядки створюються в динамічній пам'яті (про яку мова буде йти в розділі 4). З типом **PChar** сумісний будь-який символьний масив, ліва межа якого дорівнює 0. Більш того, ASCIIZ-рядки індексуються з відліком значень індексу від 0. Рядкові константи також можуть записуватися в змінні типу **PChar**. Для виводу ASCIIZ-рядків можуть бути застосовані звичайні оператори **Write** і **WriteLn**. Для забезпечення можливості оперування даними типу **PChar** необхідно включити розширений синтаксис, що здійснюється директивою **{$X+}** чи опцією Options/Compiler/Extended syntax.

Про особливості роботи з даними типу **PChar** йдеться мова наприкінці підрозділу 4.1.

```
{Приклад 3.11}
{Завдання і вивід ASCIIZ-рядка. Робота з
 символьним масивом з нульовою базою}
{$X+}                          {Вмикання розширеного синтаксису}
const
  CharsArray:array[0..20] of Char='Blez Pascal'; {Симв. масив}
var
  ASCIIZ:PChar;
begin
    ASCIIZ:=CharsArray;          {Посилання на символьний масив}
    WriteLn(ASCIIZ);                          {Друк ASCIIZ-рядка}
    ASCIIZ[4]:=#0;{Завдання замикаюч. символу – зміна довжини}
        {Тепер у CharsArray тільки 4 символа: 'B','l','e','z'}
    WriteLn(CharsArray)    {Можна друкувати і символьний масив}
end.
```

У програмі символьний масив оголошений у вигляді типізованої константи. Оскільки під нього відведений 21 байт, його вміст

потрібно було б задавати рядком з 21 символу; однак у TP 7.0 рядкова константа, довжина якої менше кількості елементів символьного масиву, також записується в символьний масив з автоматичним заповненням кінця масиву символом #0. Слід зазначити, що в наведеній програмі здійснюється корекція змінної ASCIIZ, що призводить до зміни вмісту масиву CharsArray, бо змінна ASCIIZ після присвоювання ASCIIZ:=CharsArray просто посилається на цей масив і оперує областю пам'яті, яку він займає. Тому оператор ASCIIZ[4]:=#0 змінює довжину рядка CharsArray до чотирьох символів.

ASCIIZ-рядок може бути оголошений типізованою константою, але її довжина в цьому випадку не може перевершувати 255.

Наприклад:

```
const
  PChars:PChar='Turbo Pascal 7.0';
```

До даних типу **PChar** за допомогою операції додавання (+) можна додавати цілі значення, що відповідає зсуву початку ASCIIZ-рядка на зазначене число позицій. Аналогічно виконується операція віднімання (–). Введена також операція віднімання двох ASCIIZ-рядків, результатом якої є цілочислове значення, що відповідає зсуву в пам'яті початку одного ASCIIZ-рядка стосовно початку іншого.

```
{Приклад 3.12}
{Приклади операцій з ASCIIZ-рядками}
{$X+}                          {Вмикання розширеного синтаксису}
const
   ASCIIZ_1:PChar='Turbo Pascal';
   ASCIIZ_2:PChar='Турбо Паскаль';
var
   ASCIIZ_3:PChar;
   n:LongInt;
begin
   ASCIIZ_3:=ASCIIZ_1+6;                      {ASCIIZ_3='Pascal'}
   WriteLn(ASCIIZ_3);
   WriteLn(ASCIIZ_2+6);                {Друкується слово Паскаль}
   ASCIIZ_3:='Рядкова константа';             {Запис рядкової...}
   WriteLn(ASCIIZ_3);               {...константи в ASCIIZ-рядок}
   n:=ASCIIZ_2-ASCIIZ_1;     {Взаємний зсув у пам'яті двох ...}
   WriteLn(n)                      {... ASCIIZ-рядків дорівнює n}
end.
```

Основні засоби для роботи з ASCIIZ-рядками зосереджені в модулі **Strings** (див. розділ 12).

3.3. Множини

3.3.1. Оголошення множин

У ТР існує поняття *множини*, що має смисл якогось «зібрання» елементів одного і того ж базового типу. Базовий тип визначає перелік всіх елементів, які взагалі можуть міститися в даній множині. Як базовий тип може виступати будь-який простий порядковий тип. Наприклад, дійсні числа (**Real** не порядковий тип) і рядки (не простий і не порядковий тип) не можуть бути елементами множини. Розмір множини в ТР завжди обмежений деякою гранично припустимою кількістю елементів. В множинах допускаються тільки такі елементи, порядкові значення яких не виходять за межи 0..255. Для цілочислових множин це означає, що в них можуть бути присутніми тільки числа від 0 до 255. Від'ємні елементи множин у ТР неприйнятні. Тому базовими типами не можуть бути типи **SortInt**, **Integer** і **LongInt**. Якщо ж необхідна множина цілочислових об'єктів, то базовий тип повинен бути оголошеним як відрізок типу **Byte**. Для множин, що містять символи, подібні утруднення відсутні, оскільки базовим типом для них є **Char** (а в ньому 256 значень з порядковими номерами від 0 до 255).

У математиці для позначення множин використовуються фігурні дужки (наприклад, {2, 4, 6}), у ТР – квадратні (наприклад, [2, 4, 6]) чи пари символів **(.** і **.)**. Як і в математиці, в ТР порядок елементів у множині байдуже який. Так, записавши [4,2,6] чи [2,4,6], ми будемо мати справу з тією самою множиною. Більш того, багаторазове повторення того самого елемента не змінює множину (зазначену множину можна записати і так: [2,4,2,6]).

За формою запису оголошення змінної типу «множина» подібно з оголошенням одновимірного масиву

var
ім'я**: set of** тип**;**

Наприклад, оголошення змінної characters, розглянутій як множина з базовим типом **Char**, має вигляд

```
var
  characters: set of Char;
```

На відміну від елементів масиву, елементи множини не упорядковані і не мають індексації.

Можна спочатку оголосити тип множини, а потім використовувати його для оголошення змінних:

```
type
     t_CharSet=set of Char;
var
     CharSet1,CharSet2:t_CharSet;
```

Досить часто як базовий тип множини використовується перелічуваний тип чи деякий його діапазон:

```
type
  week_days=(Monday,Tuesday,Wednesday,Thursday,Friday);
var
   Workdays:set of week_days;
   symbols:set of 'A'..'Z';
```

Відзначимо, що оголошення змінної-множини не присвоює їй значення.

3.3.2. Побудова множини

Щоб у множині з'явилися елементи, необхідно виконати оператор присвоювання, у лівій частині якого стоїть ім'я змінної-множини, а в правій – конструктор множини чи деякий вираз над множинами.

Конструктор множини — це перелік елементів у квадратних дужках (при цьому елементи розділяються комами). Як елементи можуть бути присутніми діапазони значень:

```
Workdays:=[Monday,Wednesday,Thursday];
symbols:=['C','T'..'W','A'];
```

Треба пам'ятати, що при завданні множини порядок його елементів може бути призвільним, але при завданні відрізка такий порядок важливий. Тому оператор

```
symbols:=['Z'..'F']
```

не помістить у множину symbols елементи, бо в ньому відрізок заданий у зворотному порядку щодо його оголошення.

Множина, у якій немає елементів, називається *порожньою* (або інакше *нуль-множиною*). Вона в ТР позначається квадратними дужками, між якими не зазначені елементи:

```
Workdays:=[];
```

Множину можна оголосити у вигляді типізованої константи, для чого в описі після знаку рівності треба вказати конструктор множини. Наприклад, оператор

```
const   RusLetters:set of ['А'..'Я']=
        ['А','Е','И','О','У','Э','Ю','Я','Ы'];
```

описує множину, елементами якої можуть бути великі російські літери, із записом у неї початкового значення, що є множиною великих голосних російських букв.

Конструюючи множини, можна використовувати і змінні за умови, що їхні поточні значення потрапляють у діапазон базового типу множини. Так, якщо ch1 і ch2 мають тип **Char**, то припустима така послідовність операторів:

```
ch1:='S';ch2:='A';symbols:=[ch1,'F',ch2];
```

Тут результуючою множиною буде ['A','F','S'].

Результат обчислення множинного виразу повинний бути сумісним за присвоюванням зі змінною, що стоїть у лівій частині оператора присвоювання.

Елементи множини не можна вводити і виводити. Для організації введення/виводу елементів множини треба використовувати допоміжні змінні, як це зроблено в програмі, наведеній у прикладі 3.14. У той же час можна використовувати множини як елементи типізованих файлів.

3.3.3. Дії над множинами

3.3.3.1. Об'єднання, перетинання і різниця множин

Над множинами здійсненні, насамперед, слідуючи три операції: об'єднання (+), перетинання (*) і різниця (–).

Об'єднання двох множин A і B (A+B) — це нова множина, що складається з елементів, які належать множині A чи B або тій та іншій одночасно:

```
var
   CharSet1,CharSet2,CharSet3: set of Char;
begin
   CharSet1:=['a','x','e'];
   CharSet2:=['u','x'];
   CharSet3:=CharSet1+CharSet2+['s','w'];
end.   {Тут CharSet3=['a','e','s','u','x']}
```

Перетинання двох множин A і B (A*B) — це множина, що складається з елементів, які одночасно належать множинам A і B:

```
CharSet3:=CharSet1*CharSet2; {CharSet3=['x']}
CharSet1:=CharSet1*['n','d'];    {CharSet1=[]}
```

Різниця двох множин A і B (A–B) — це нова множина, що складається з елементів множини A, які не увійшли в множину B:

```
CharSet1:=['a','e','t'];
CharSet2:=CharSet1-['e'];     {CharSet2=['a','t']}
CharSet3:=['s','e','t']-CharSet1;{CharSet3=['s']}
CharSet2:=CharSet1-['s','e','t'];{CharSet2=['a']}
```

Маніпулюючи операціями над множинами, можна додавати елементи до множин чи видаляти їх.

Для вставки і видалення елементів при роботі з множинами в TP 7.0 введені дві процедури:

Include(ім'я_множини**,** елемент**)** – вставити елемент;

Exclude(ім'я_множини**,** елемент**)** – видалити елемент.

Перша з них забезпечує додавання одного елемента в зазначену множину, а друга – видалення:

```
Include(CharSet1,'g'); {Те ж, що і CharSet:=CharSet+['g']}
Exclude(CharSet1,'t'); {Те ж, що і CharSet:=CharSet-['t']}
```

Ці процедури дають більш ефективний код у порівнянні з операціями «+» і «–».

3.3.3.2. Інші операції над множинами. Пріоритет операцій

Над множинами можна виконувати чотири операції порівняння: =, <>, <=, >=.

Дві множини A і B *рівні* (A=B), якщо кожен елемент множини A є також елементом множини B і навпаки.

Дві множини A і B *не рівні* (A<>B), якщо вони відрізняються хоча б одним елементом.

Множина A є *підмножиною* множини B (A<=B, чи B>=A), якщо кожен елемент з A є присутнім у B.

Є також можливість з'ясувати, чи належить даний елемент деякій множині. Для цього служить операція **in**. Нехай A – множина елементів деякого базового типу, а *x* – змінна (константа, вираз) цього типу. Тоді вираз *x in A* є вірним, якщо значення *x* є елементом множини A.

Всі операції порівняння множин, а також операція **in** виробляють значення **True** чи **False**.

У складних виразах над множинами операції виконуються у відповідності з такою системою пріоритетів:

1 (вищий пріоритет)	*
2	+, –
3 (нижчий пріоритет)	=, <>, <=, >=, **in**

Операції одного пріоритетного рівня виконуються зліва направо (якщо не використані дужки).

У наведеній нижче програмі показано, як описується типізована константа-множина, проілюстровані перевірка приналежності множині і застосування операції об'єднання множин.

```
{Приклад 3.13}
{Вводяться два рядки. Чи є результат конкатенації
 цих рядків ідентифікатором (ім'ям)? Ім'я може містити
 не більше 127 символів, починатися з  літери чи символу
 підкреслення і містити літери, цифри і символ підкреслення }
type
   t_set=set of Char;
const
   Letters:t_set=['a'..'z','A'..'Z','_'];
   Ciphers:t_set=['0'..'9'];
var
   s1,s2,s:string;
   i:Byte;
   flag:Boolean;
begin
   WriteLn('Введіть 2 рядки');
   ReadLn(s1);ReadLn(s2);
   s:=s1+s2;
   flag:=True;
    {Перевірка першого символу і довжини рядка на допустимість}
   if not (s[1] in Letters) or (Length(s)>127)
   then flag:=False;
            {Перевірка допустимості всіх інших символів рядка}
   i:=2;
   while (i<=Length(s))and flag do begin
     if not(s[i] in Letters+Ciphers) then begin
        flag:=False;Break;
     end;
     i:=i+1
   end;
   if flag then WriteLn('Так')
   else  WriteLn('Hi')
end.
```

Використання операцій перетинання і різниці множин, а також введення/виведення вмісту множини ілюструє наведена нижче програма.

```
{Приклад 3.14}
{Лотерея "5 з 36": вгадати 5 загаданих чисел
 від 1 до 36. При програші надрукувати, які числа
 вгадані, які не вгадані, які введені невірно}
type
  t_1_36=1..36;                  {Діапазон чисел, що загадуються}
var
   number:t_1_36;
   n:0..5;
   comput,                                {Множина загаданих чисел}
   player,                          {Множина чисел, названих гравцем}
   set1:set of t_1_36;                          {Допоміжна множина}
begin
  n:=0;
  Randomize;
  comput:=[];                                          {Порожня множина}
  repeat                {Завдання 5 випадкових чисел від 1 до 36}
    number:=Random(36)+1;
    if not(number in comput) then begin           {Якщо числа...}
      n:=n+1;                {...numb немає в comput, додати його}
      Include(comput,number)      {Можна comput:=comput+[number]}
    end
  until n=5;
  player:=[];
  WriteLn('Введіть через <Enter> 5 цілих чисел від 1 до 36');
  for n:=1 to 5 do begin
    ReadLn(number);      {Ці два оператори "вводять" елемент...}
    Include(player,number)                              {... у множину}
  end;      {Нижче ілюструється операція порівняння двох множин}
  if player=comput then WriteLn('Поздоровляю, Ви виграли')
  else begin
    WriteLn('На жаль, Ви програли');
    for n:=1 to 3 do begin
      case n of
       1:begin Write('Угадані числа: ');
               set1:=comput*player          {Перетинання множин}
         end;
       2:begin Write('Не вгадані числа: ');
               set1:=comput-player                {Різниця множин}
         end;
       3:begin Write('Помилкові числа: ');
               set1:=player-comput                {Різниця множин}
         end;
      end;
```

```
      for number:=1 to 36 do              {Вивід вмісту множини}
        if number in set1 then Write(number:3);
      WriteLn;
    end
  end
end.
```

У даній програмі показано, що замість введення даних у множину, достатньо здійснювати введення в допоміжну змінну з наступним включенням введеного значення в множину (за допомогою **Include**). Вивід вмісту множини організують у циклі за всіма значеннями базового типу з друком тільки тих з них, що належать множині.

3.4. Записи

3.4.1. Звичайні записи

Розглянемо таку задачу. Є інформація про 100 студентів, що включає прізвища і середній бал сесії. Необхідно надрукувати прізвища студентів, успішність яких вище середньої.

У цій задачі інформація про кожного зі студентів складається з даних двох типів – рядкового (прізвище) і числового (середній бал). При написанні програми можна скористатися двома одновимірними масивами, які повинні оброблятися спільно. Це незручно, особливо у випадках, коли подібних характеристик не дві, а значно більше. Багатовимірні масиви тут допомогти не можуть, оскільки кожна з характеристик має свій тип, а масиви поєднують у собі тільки однотипні елементи.

У ТР введений спеціальний комбінований тип даних, який дозволяє поєднувати відразу кілька типів. Такий тип даних називають записом. Формат опису змінної типу «запис» такий:

```
var
  ім'я_запису:record
            ім'я1:тип1;
            ім'я2:тип2:
              . . .
            ім'яN:типN;
  end;
```

Частини запису, призначені для безпосереднього збереження інформації («ім'я1», ..., «ім'яN»), називаються полями запису. Для

кожного з полів повинні бути зазначені ім’я і тип. Якщо кілька полів мають один тип, то їх можна оголосити спільно, перелічивши імена через кому.

Для сформульованої задачі можна, наприклад, оголосити таку змінну типу «запис»:

```
var
   stud:record
          surname:string[20];       {Прізвище}
          AverageEval:Real    {Середня оцінка}
   end;
```

Підкреслимо, що опис запису завершується службовим словом **end**, перед яким можна ставити, а можна не ставити крапку з комою.

Рекомендується спочатку ввести ім’я типу запису, яке надалі буде використовуватися при описі змінних:

```
type
   t_stud:record
             surname:string[20];
             AverageEval:Real
   end;
var
   stud:t_stud;
```

Щоб звернутися до поля запису, необхідно не просто вказати ім’я поля, але перед ним вказати ім’я запису, з’єднавши їх крапкою:

```
ReadLn(stud.surname);
ReadLn(stud.AverageEval);
```

Альтернативою такому звертанню є оператор приєднання **with**:

with ім’я_запису **do** оператор**;**

Оператор **with** автоматично приєднує зазначене в ньому «ім’я_запису» до всіх імен, які входять в оператор, що стоїть після **with**, і збігаються з іменами полів запису. До імен, що не збігаються з іменами полів запису, приєднання не проводиться. Якщо приєднання потрібно здійснити в декількох операторах, то їх за допомогою **begin-end** поєднують у складений оператор. Наприклад, наведені вище оператори можна записати так:

```
with stud do begin
   ReadLn(surname);
   ReadLn(AverageEval)
end;
```

При оголошенні типізованої константи записного типу її значення задається в такий спосіб: у загальних круглих дужках перелічують імена полів, після кожного з яких через двокрапку вказується початкове значення; роздільником є символ «крапка з комою»:

```
  const
     stud1:t_stud=(surname:'Іванов';AverageEval:0.0);
```

Записи можуть бути елементами масиву, як це має місце в програмі, що наводиться нижче.

```
{Приклад 3.15}
{Надрукувати прізвища тих з n студентів (n<=100),
 чий середній бал вище загального середнього бала}
type
   stud_rec=record
               surname:string[20];
               AverageEval:Real
   end;
var
   stud_array:array[1..100] of stud_rec;
   sum,ArithmeticMean:Real;
   i,n:Integer;
begin
   Write('Введи кількість студентів: n=');
   ReadLn(n);
   sum:=0;
   for i:=1 to n do
      with stud_array[i] do begin
         Write('Введи прізвище: ');ReadLn(surname);
         Write('Введи середній бал: ');ReadLn(AverageEval);
         sum:=sum+AverageEval
      end;
   ArithmeticMean:=sum/n;
   for i:=1 to n do
      if stud_array[i].AverageEval>ArithmeticMean
         then WriteLn(stud_array[i].surname)
end.
```

У програмі показано, як можна використовувати оператор **with** (до імені sum приєднання не буде здійснюватися, бо це не ім'я поля запису stud_array[i]) і як звертатися до поля запису безпосередньо (у другому операторі циклу).

Поля запису можуть мати будь-який тип. Наприклад, полем запису може бути масив

```
type
   session_rec=record
                  AverageEval:Real;
                  Eval:array[1..5] of 2..5
   end;
```

чи запис:

```
type
   stud_rec1=record
                surname:string[20];
                session:session_rec
   end;
var
   report:array[1..100] of stud_rec1;
```

В останньому випадку для звертання до полів запису доводиться будувати цілий ланцюжок приєднань. Можна також вказати через кому список імен, що приєднуються, в операторі **with**. При цьому приєднання починається з імені, яке стоїть правіше в списку. Наприклад, такі чотири оператори еквівалентні:

1) `report[17].session.Eval[3]:=4;`
2) `with report[17], session do Eval[3]:=4;`
3)
```
with report[17] do
      with session do Eval[3]:=4;
```
4) `with report[17].session do Eval[3]:=4;`

Можливість використання записів як полів інших записів дозволяє організовувати в програмі оперування складними деревоподібними структурами різнорідних даних.

3.4.2. Записи з варіантами

Нехай тепер необхідно організувати обробку інформації про книги (автори, назва книги, місце видання, назва видавництва, рік видання) і журнальні статті (автори, назва статті, журнал, номер, рік видання). Щоб в одному масиві записів зосередити інформацію про такі різнорідні записи, потрібно описати в одному запису всі можливі поля і якимось чином обмежити доступ до тих полів, які не використовуються у конкретному запису. Щоб позбутися цього, у ТР є можливість використання записів з варіантами, ідея яких складається у визначенні в рамках однієї структури декількох різних (альтернативних) записів.

Запис з варіантами складається з двох частин – фіксованої, в якій перелічуються загальні для всіх альтернатив поля, і мінливої (варіантної), структура якої змінюється залежно від значення особливого поля, яке іноді називають полем тега (ознаки). Для поставленої вище задачі можна запропонувати таку структуру запису:

```
type
   t_catalog=record
             author:string[50];
             title :string[100];
             year  :Integer;
             case tag:Byte of
                0  :(town:string[15];
                     publishing:string[20]);
                1  :(journal:string[50];num:Byte)
   end;
var
   catalog:array[1..100] of t_catalog;
```

Особливості записів з варіантами:

- варіантна частина може бути тільки одна, і вона повинна бути останньою;
- варіантна частина є подібною оператору **case**, в якого відсутнє службове слово **end;** як **end** для **case** використовується **end** для **record**;
- полем тега є ім’я (власне тег) і тип, що стоять за **case**; поле тега може складатися тільки з типу без зазначення імені;
- тег може мати тільки дискретний тип;
- тег входить у фіксовану частину запису і завершує її;
- опис полів, що відносяться до окремих значень тега, беруть у круглі дужки;
- якщо варіант для якого-небудь значення тега відсутній, то він задається порожнім полем, укладеним у круглі дужки;
- якщо якесь з полів у варіантній частині теж варіантний запис, то воно повинно бути останньою частиною варіанта;
- якщо типізована константа оголошена варіантним записом, то при задані її значення розглядають тільки один варіант.

Варіантні записи, полегшуючи оперування даними, у той же час збільшують обсяг пам’яті, що витрачається, оскільки під такого роду змінні завжди виділяється пам’ять, яка необхідна для збереження

самого довгого запису. Контроль над тим, який з варіантів реалізований у даній змінній, цілком покладається на програміста (можна, наприклад, записати в деяку змінну catalog[i] інформацію про книгу, а обробляти її як журнальну статтю). Тому при роботі з варіантними записами треба завжди враховувати значення тега.

```
{Приклад 3.16}
{Є каталог на n книг і журнальних статей (n<=100).
 Вивести інформацію про публікації, видані після
 1990 р. (окремо книги і статті)}
type
   t_publication=record
       author:string[50];
       title:string[100];
       year:Integer;
       case key:Char of
       'B','b':(town,publishing:string[20]);     {книга - book}
       'A','a':(journal:string[50];num:Byte);{стаття - article}
   end;
var
   catalog:array[1..100] of t_publication;
   n,i:Integer;
begin
   Write('Введи кількість записів: n=');ReadLn(n);
     {У програмі відсутнє введення даних у масив catalog.
      Воно залишено на самостійне проопрацювання в зв'язку
      з тим, що при введенні з клавіатури цей фрагмент
      програми є досить довгим.
      При перевірці програми не забути організувати введення! }
   for i:=1 to n do                  {Друк інформації про книги}
      with catalog[i] do
         if (year>1990)and(key in['B','b']) then
            WriteLn(author,'.',title,'.',town,': ',
                    publishing,'.-',year);
   for i:=1 to n do              {Виведення інформації про статті}
      with catalog[i] do
         if (year>1990)and(key in['A','a']) then
            WriteLn(author,'.',title,'//',journal,'.',
                    year,'. N ',num);
end.
```

У програмі поле key використовується для розпізнання запису: що це, книга чи стаття?

Природно, вводити подібного роду дані з клавіатури і виводити на екран недоцільно, а при великих значеннях n неможливо. У цьому випадку на допомогу приходять файли.

3.5. Файли

3.5.1. Текстові файли

Введення даних з клавіатури і вивід їх на екран не завжди зручні: найчастіше більш доцільним є використання підготовленого заздалегідь набору даних, який зберігається у вигляді файла, і вивід даних у файл з метою подальшого їхнього використання. TP дозволяє оперувати файлами, одним з видів яких є текстові файли, що складаються із символьних рядків змінної довжини, що завершуються спеціальною комбінацією, яка зветься «кінець рядка». Комбінація «кінець рядка» складається з двох символів: «переведення каретки» (#13) і «переведення рядка» (#10). Завершується текстовий файл символом «кінець файла» (#26). Такий файл можна підготувати будь-яким текстовим редактором, у тому числі редактором, вбудованим в інтегроване середовище TP.

Опис текстового файла здійснюється оголошенням змінної типу **Text**:

var файлова_змінна**:Text;**

Особливістю файлових змінних, як текстових, так і всіх інших файлів, є те, що з ними не може працювати оператор присвоювання.

Для роботи з файлом одного оголошення недостатньо, оскільки оголошена файлова змінна не є самим файлом або хоча б його ім'ям, яке відоме операційній системі. Це лише внутрішнє ім'я файла, з яким працює програма. Але програмі повинні бути відомі місце розташування файла (ім'я дисковода і каталог), а також ім'я і розширення, з якими він записаний на диску. Ці відомості повідомляються за допомогою оператора **Assign** (призначити), що має такий формат:

Assign(файлова_змінна**,** зовнішнє_ім'я_файла**);**

Цей оператор зв'язує ім'я файлової змінної, що фігурує в програмі, із зовнішнім ім'ям файла, яке задається у вигляді рядка (константи, змінної, виразу). Наприклад, якщо в програмі є опис вигляду

```
var f:Text;
```

то оператор

```
Assign(f,'a.txt');
```

зв’язує файлову змінну *f* з файлом a.txt, розташованим у поточному каталозі активного диска.

При зазначенні порожнього рядка як параметра «зовнішнє_ім’я_файла» файл зв’язується зі стандартними пристроями введення/виводу даних (клавіатурою при введенні й екраном при виводі). Наприклад, оператор Assign(f,’’) зв’язує файл *f* з клавіатурою чи екраном залежно від напрямку обміну даними.

Для роботи з вмістом файла недостатньо зв’язати внутрішнє ім’я файла із зовнішнім: після виконання оператора **Assign** і до першої операції читання з файла чи запису у файл останній повинен бути відкритий. Для читання даних з текстового файла він відкривається оператором

Reset(ім’я_файлової_змінної**);**

а для запису даних у файл відкриття здійснюють оператором

Rewrite(ім’я_файлової_змінної**);**

Наприклад, для змінної *f* ці оператори виглядають так: Reset(f) і Rewrite(f). Виконання кожного з цих операторів позиціонує файл на позицію його першого символу. Тому відкриття існуючого файла за допомогою **Rewrite** забезпечує його повне відновлення. Треба пам’ятати, що текстовий файл може бути відкритий тільки для читання чи тільки для запису.

По закінченні роботи з файлом його закривають оператором

Close(ім’я_файлової_змінної**),**

який не є обов’язковим, бо при завершенні виконання програми усі файли закриваються автоматично. Повторне відкриття файла без його закриття супроводжується попереднім автоматичним закриттям.

Для текстових файлів введена ще одна процедура відкриття для запису – **Append**, яка має той же формат, що і **Rewrite**. Відмінність її в тому, що вона працює з існуючим файлом і позиціонує файл на його кінець, у результаті чого файл не оновлюється, а здійснюється дозапис даних у його кінець.

Обмін даними між програмою і файлом провадиться не прямо, а через спеціальний системний буфер розміром 128 байт. При заповненні буфера йде його автоматичне звільнення з занесенням даних у файл. Оператор **Close**, крім інших дій, переписує з буфера залишок даних.

Якщо файл не закритий, то, звернувшись до процедури

Flush(ім'я_файлової_змінної**),**

можна примусово переписати вміст буфера виводу в файл.

Для текстового файла можна створити буфер у самій програмі, а не в системній пам'яті, для чого досить скористатися процедурою

SetTextBuf(ім'я_файлової_змінної**,** ім'я_буфера**,** розмір_буфера**).**

Другий параметр у ній — це ім'я будь-якої змінної, під яку відведений обсяг пам'яті, не менший значення третього параметра (у байтах). Наприклад,

```
SetTextBuf(f,buf,200);
```

де buf – масив, який має такий опис:

```
var buf:array[1..200] of Char;
```

Третій параметр у цій процедурі можна пропустити, встановивши розмір буфера за замовчуванням (128 байт). При встановленні буфера файл не повинен бути відкритим.

Читання з текстового файла здійснюється операторами **Read** і **ReadLn**, в яких як перший параметр вказують ім'я файлової змінної, а далі через кому перелічують змінні, в які здійснюється читання даних з файла. У текстовому файлі дані зберігаються в рядковому вигляді; однак, якщо елемент даних може бути перетворений у число, це перетворення здійснюється автоматично при введенні в числові змінні. Елементи числових даних у рядках текстового файла розділяються прогалинами чи символами табуляції (клавіша <Tab>). Якщо рядок файла вичерпаний, а в операторі **Read** не вичерпався список введення, який складається з числових чи символьних змінних, то введення продовжується з наступного рядка. При введенні даних з текстового файла в символьні змінні елементи даних не розділяються. Якщо в списку даних після числової змінної йде рядкова, то прогалина, яка іде після числового значення у файлі, зчитується в рядок (це ж справедливо і при зчитуванні в символьну змінну). Треба враховувати, що при зчитуванні даних у рядок прогалини сприймаються як значущі символи, і в рядок потрапляють усі дані до комбінації «кінець рядка» чи ж до вичерпання максимальної довжини рядка. Тому рядкові дані повинні розташовуватися в текстовому файлі або наприкінці рядка, або в окремих рядках і читатися оператором **ReadLn**.

Відмінність операторів **Read** і **ReadLn** при читанні з текстових файлів полягає в тому, що оператор **ReadLn**, помістивши значення в останню змінну списку введення, переходить на початок наступного

рядка, не зчитуючи в рядку дані, що залишилися, а оператор **Read** зостається готовим зчитувати дані з наступної позиції поточного рядка. Так, наприклад, якщо в текстовому файлі *f* є два рядки

1 -2
4

то два оператори Read(f,m);Read(f,n); помістять у цілочислові змінні *m* і *n* відповідно значення 1 і -2, а два оператори ReadLn(f,m); ReadLn(f,n); зчитають значення 1 і 4.

Якщо при читанні з файла буде прочитаний символ кінця файла, то виникне помилка часу виконання, для виключення якої необхідно перед читанням перевіряти, чи немає позиціювання на кінець файла. Для цього в ТР введена функція

Eof(ім'я_файлової_змінної**),**

що повертає значення **True**, якщо в момент звертання до неї виявлений кінець файла, і **False** у іншому випадку.

При виводі даних у текстовий файл використовуються оператори **Write** і **WriteLn**, першим параметром яких є ім'я файла, а далі через кому іде список значень, що виводяться (констант, змінних, виразів). Відмінність їх у тому, що по закінченні виводу оператор **WriteLn** позиціонує файл на наступний рядок, забезпечуючи наступний вивід з нового рядка. Якщо декілька даних виводиться в один рядок текстового файла, то вони ніяк не розділяються. Дані, які не є рядковими (числові, булевські), автоматично перетворюються у рядковий вигляд.

```
{Приклад 3.17}
{У текстовому файлі f.txt через прогалину і <Enter> записані
 цілі числа. Переписати у файл f1.txt з файла f.txt усі
 числа за винятком максимальних (у припущенні,
 що їх може бути декілька}
var
  f,f1:Text;                          {Оголошення файлових змінних}
  a,max:LongInt;                {Число та максимальне з усіх чисел}
  flag:Boolean;
begin
  Assign(f,'f.txt');     {Зв'язування файлової змінної з файлом}
  Reset(f);                        {Відкриття файла f для читання}
  flag:=True;
  while not Eof(f) do begin                {Читати до кінця файла}
     Read(f,a);                        {Читання з елемента файла f}
     if flag or(a>max) then max:=a;
     flag:=False;
  end;
```

```
  Assign(f1,'f1.txt');  {Зв'язування файлової змінної з файлом}
  Rewrite(f1);                    {Відкриття файла f1 для запису}
  Reset(f); {Повторне відкриття файла f для повторного читання}
  while not Eof(f) do begin
     Read(f,a);
     if a<>max then WriteLn(f1,a);  {При виводі пропускаємо...}
  end;                                 {...максимальні елементи}
  Close(f1);Close(f)                              {Закриття файлів}
end.
```

У наведеній вище програмі замість функції **Eof** краще використовувати функцію **SeekEof**, оскільки у випадку наявності наприкінці файла f.txt порожніх рядків чи прогалин вони сприймаються як значення 0, у результаті чого цей 0 вже як число може потрапити у файл f1.txt.

Для текстових файлів введені ще три функції:

Eoln(ім'я_файла**), SeekEoln(**ім'я_файла**), SeekEof(**ім'я_файла**).**

Перша з них перевіряє, досягнутий кінець рядка (**True**) чи не досягнутий (**False**) у файлі, ім'я якого задане як параметр; друга, працюючи аналогічно першій, пропускає при аналізі прогалини і символи табуляції; третя функція працює аналогічно функції **Eof**, але при аналізі пропускає прогалини і символи табуляції.

Особливістю текстових файлів є те, що вони є файлами послідовного доступу: не можна прочитати який-небудь елемент текстового файла, не прочитавши всі попередні елементи. Аналогічно не можна записувати інформацію в текстовий файл довільним чином, писати в нього можна тільки послідовно.

3.5.2. Обробка помилок введення/виводу

За замовчуванням у TP здійснюється автоматична перевірка на наявність помилок введення/виводу з видачею діагностичного повідомлення і припиненням виконання програми у випадку їхнього виникнення. Цей контроль можна включати або відключати, для чого служить директива компіляції **$I**:

{$I+} – автоматичний контроль введення/виводу включений;

{$I–} – автоматичний контроль введення/виводу виключений.

Для обробки помилок введення/виводу користувачем директиву **$I** необхідно встановити в пасивний стан **{$I–}** і відразу ж після виконання операції введення/виводу звернутися до функції **IOResult**, яка не має параметрів. Якщо помилки введення/виводу нема, то

функція повертає нульове значення; при виникненні ж помилки повертається цілочислове значення – код помилки, і програма продовжує виконуватися. Коди помилок введення/виводу такі:

100 – помилка читання з диска;
101 – помилка запису на диск;
102 – файлу не привласнене ім'я;
103 – файл не відкритий;
104 – файл не відкритий для введення;
105 – файл не відкритий для виводу;
106 – невірний числовий формат.

Код помилки може бути проаналізований користувачем для організації відповідної реакції.

Треба пам'ятати, що при **{$I–}** у випадку виникнення помилок введення/виводу всі наступні операції по роботі з файлами ігноруються до звертання до функції **IOResult**. Звертання до функції **IOResult** без виконання операцій введення/виводу, у тому числі повторне для однієї і тієї ж операції, дає значення 0.

Після фрагмента програми, у якому можливі помилки введення/виводу, необхідно відновити стан директиви: **{$I+}**.

```
{Приклад 3.18}
{Фрагмент програми з відкриттям текстового файла для
 читання при введенні його імені з клавіатури}
var
   f:Text;
   NameOfFile: string;
   code:Integer;
begin
{$I-}                          {Вимикання автоматичного контролю}
   repeat
     Write('Введи ім''я файла:');
     ReadLn(NameOfFile);
     Assign(f,NameOfFile);
     Reset(f);
     code:=IOResult;
     if code<>0 then
     WriteLn('Файл відсутній чи не відкривається для введення')
   until code=0;
{$I+}                           {Вмикання автоматичного контролю}
   {Продовження програми}
end.
```

3.5.3. Логічні пристрої і стандартні файли введення/виводу

У TP усі стандартні засоби комп'ютера (клавіатура, екран, принтер, комунікаційні канали введення/виводу) розглядаються як потенційні джерела текстової інформації. Вони визначаються спеціальними іменами, які називаються логічними пристроями:

con – консоль (клавіатура для читання даних і екран дисплея для виводу);

prn – принтер;

lpt1, lpt2, lpt3, lpt4 – логічні імена принтерів, якщо їх декілька; імена **prn** і **lpt1** за замовчуванням є синонімами;

aux – комунікаційний канал для зв'язку з іншими комп'ютерами;

com1, com2, com3 – імена комунікаційних каналів, якщо їх декілька; за замовчуванням **aux** і **com1** – синоніми;

nul – «порожній» пристрій; він використовується в режимі налагодження і трактується як пристрій необмеженої місткості.

Зв'язування логічного пристрою з ім'ям файлової змінної здійснюється звичайним чином, ніби логічний пристрій є зовнішнім ім'ям текстового файла. При цьому необхідно відкрити файл відповідно до тих обмежень для операторів **Reset**, **Rewrite** і **Append**, що встановлені для їхнього використання при відкритті текстових файлів.

```
{Приклад 3.19}
{Використовуючи логічні пристрої, здійснювати введення чисел
 до натискання Ctrl/Z. Чи правда, що сума введених чисел
 менше добутку? Організувати вивід на екран і принтер}
var
   f1,f2,f3:Text;
   a,sum,mult:Real;
begin
   Assign(f1,'con');
   Reset(f1);           {Забезпечення введення з клавіатури}
   Assign(f2,'con');
   Rewrite(f2);              {Забезпечення виводу на екран}
   Assign(f3,'prn');
   Rewrite(f3);           {Забезпечення виводу на принтер}
   sum:=0;mult:=1;
   WriteLn('Вводьте числа до натискання Ctrl/Z');
   while not Eof(f1) do begin  {Читати до натискання Ctrl/Z}
      ReadLn(f1,a);                    {Введення з клавіатури}
      sum:=sum+a;mult:=mult*a;
   end;
```

```
   WriteLn(f2,sum<mult);                          {Вивід на екран}
{$I-}
   WriteLn('Підключить принтер');
   repeat
      Write(#7);                        {Видача звукового сигналу}
      WriteLn(f3,sum<mult);                      {Вивід на принтер}
   until IOResult=0;              {Повторювати, поки не буде...}
                             {...забезпечений вивід на принтер}
{$I+}
   Close(f3);Close(f2);Close(f1)
end.                        {У програмі виводиться TRUE або FALSE}
```

У будь-якій TP-програмі вважаються відомими два текстових файли з іменами **Input** і **Output**. Перший з них за замовчуванням зв'язаний з клавіатурою (стандартний файл введення), а другий – з екраном (стандартний файл виводу). Стандартні файли **Input** і **Output** відкривати не потрібно – вони завжди вважаються відкритими. При введенні/виводі з використанням цих файлів їх імена можна пропустити, і ми приходимо до вже звичних операторів введення/виводу (так, оператори ReadLn(a) і ReadLn(Input,a) еквівалентні). За допомогою оператора **Assign** імена **Input** і **Output** можна перепризначити, зв'язавши їх з дисковими файлами, але тоді їх перед використанням необхідно відкрити. Якщо імена **Input** і **Output** були зв'язані з дисковими файлами і потрібно повернути їхнє стандартне призначення, то за допомогою оператора **Assign** їх потрібно зв'язати з логічним пристроєм **con**, після чого ці файли повинні бути відкриті – перший для читання, другий для запису.

3.5.4. Типізовані файли

Більш характерними для TP є типізовані файли, чи файли довільного доступу, фундаментальною властивістю яких є те, що ця структура даних являє собою послідовність компонентів одного й того ж типу. Описують подібний файл словосполученням **file of** з наступною вказівкою типу компонентів файла, число яких (довжина файла) не фіксується:

var ім'я_файла**: file of** тип_компонентів**.**

Оскільки відомий тип компонентів файла, а отже, і об'єм пам'яті, що відводиться під кожний з них, то можна розрахувати позицію кожного з компонентів усередині файла, що дозволяє організувати

безпосередній доступ до будь-якого компонента типізованого файла. Так, наприклад, опис

```
var
    FileInt: file of Integer
```

говорить про те, що компонентами файла є дані типу **Integer**, які займають 2 байти. При цьому відпадає необхідність у спеціальному поділі окремих компонентів файла, як це має місце в текстових файлах, і можливий довільний доступ до них (у цьому відношенні, типізований файл трохи нагадує одновимірний масив).

Щоб можна було працювати з типізованим файлом, необхідно, як і у випадку текстових файлів, спочатку зв'язати ім'я файлової змінної із зовнішнім ім'ям файла (оператор **Assign**), а потім відкрити його (оператори **Reset** і **Rewrite**, але не **Append**). Оператори **Reset** і **Rewrite** відкривають файл одночасно і для читання, і для запису (а не тільки для читання чи тільки для запису, як це має місце в текстових файлах). Відмінність їх у тім, що оператор **Reset** відкриває тільки існуючий файл (якщо такого файла нема, то буде помилка часу виконання), а оператор **Rewrite** створює новий файл (якщо файл з таким ім'ям вже є, то він буде знищений і створений заново). При відкритті файла з ним зв'язується поточний покажчик файла, що позиціонується на його перший компонент. Оперувати можна тільки тим компонентом файла, на який вказує покажчик файла. При читанні чи запису компонента файла відбувається автоматичне переміщення покажчика на наступний компонент. Читання з типізованого файла здійснюється оператором **Read** (але не **ReadLn**), а запис у нього – оператором **Write** (але не **WriteLn**), що збігаються за форматом з аналогічними операторами для текстових файлів; однак у списку виводу оператора **Write** можуть бути тільки змінні. Типи компонентів файла і типи змінних у списках введення/виводу повинні бути сумісними за присвоюванням. Компонентами типізованих файлів можуть бути числові, символьні, булевські, рядкові значення, масиви, записи, але не файли чи структури з файловими компонентами.

Взнати кількість компонентів типізованого файла (розмір файла) можна, звернувшись до функції

FileSize(ім'я_файла**).**

Наприклад, якщо змінна *k* має тип **LongInt**, а *f* – файлова змінна типізованого файла, то оператор *k:=FileSize(f)*, записує в змінну *k* розмір файла *f*.

Компоненти типізованого файла нумеруються, причому нумерація провадиться, починаючи з 0 (порядковий номер останнього елемента файла на одиницю менше розміру файла). Щоб довідатися, на якому компоненті розташовується покажчик файла, використовують функцію

FilePos(ім'я_файла**).**

Положенням поточного покажчика можна керувати, для чого використовується процедура **Seek**, яка має такий формат:

Seek(ім'я_файла**,** номер_компонента**).**

Другий параметр (тип **LongInt**) задає номер компонента (відлік від 0), на який повинен переміститися покажчик файла:

Seek(f,5) – перейти до п'ятого (фактично шостого) компонента файла *f*;

Seek(f,FilePos(f)-1) – перейти до попереднього компонента;

Seek(f,FileSize(f)) – перейти на кінець файла.

Закриття типізованого файла здійснюється раніше розглянутою процедурою **Close**.

Як і для текстових файлів, можна використовувати функцію **Eof(**ім'я_файла**)**, що повертає значення **True**, якщо поточний покажчик розташований на ознаці кінця файла (тобто при виконанні рівності **FilePos(**ім'я_файла**)=FileSize(**ім'я_файла**)**).

Процедура **Seek** і функції **FilePos** і **FileSize** дозволяють легко здійснювати корекцію компонентів файла й організовувати дозапис у кінець файла зі збільшенням його розміру.

Процедура

Truncate(ім'я_файла**)**

знищує усі компоненти типізованого файла, ім'я якого зазначене як її параметр, починаючи з компонента, на якому розташований поточний покажчик. Однак знищити компонент усередині файла не можна, для цього файл повинен бути переписаний.

Текстові файли можуть бути створені текстовим редактором; типізовані файли створюються в результаті роботи якої-небудь програми, найпростіший приклад якої наводиться нижче.

```
{Приклад 3.20}
{Записати у файл прізвища і телефони 10 чоловік}
type
   t_subscriber=record
             surname:string[20];
             tel:LongInt
   end;
var
  subscriber: t_subscriber;                          {Абонент}
  f: file of t_subscriber;
  i:Integer;
begin
  Assign(f,'notebook.dat');
  Rewrite(f);                                        {Новий файл}
  for i:=1 to 10 do
    with subscriber do begin
       Write('Введіть прізвище '); ReadLn(surname);
       Write('Введіть номер телефону '); ReadLn(tel);
       Write(f,subscriber)           {Запис компонента у файл}
    end;
  Close(f)                                          {Закриття файла}
end.
```

Природно, при необхідності типізованими файлами можна оперувати, розглядаючи їх як файли послідовного доступу. Це ілюструє така програма:

```
{Приклад 3.21}
{На телефонній станції шестизначні номери, які починаються
з 12, замінюються семизначними номерами, що починаються
з 712. Внести зміни у файл notebook.dat}
type
   t_subscriber=record
             surname:string[20];
             tel:LongInt
   end;
var
   subscriber: t_subscriber;                         {Абонент}
   f: file of t_subscriber;
   s:string[7];
begin
   Assign(f,'notebook.dat'); Reset(f);          {Існуючий файл}
   while not Eof(f) do begin       {Тут файл розглядається...}
                              {...як файл послідовного доступу}
     Read(f,subscriber);           {Читання компонента з файла}
     with subscriber do begin
        Str(tel,s);
        if Copy(s,1,2)='12' then tel:=tel+7000000;
     end;
```

```
      Seek(f,FilePos(f)-1);   {При читанні покажчик змістився}
             {... Повертаємо його назад. Вже довільний доступ}
      Write(f,subscriber)            {Змінюємо компонент файла}
    end;
    Close(f)                                      {Закриття файла}
end.
```

У наведеній програмі типізований файл обробляється і як файл послідовного доступу, і як файл довільного доступу.

Як компоненти файлів можуть бути не тільки звичайні записи, але і варіантні. У наступному прикладі показано, як можна сформувати подібний файл.

```
{Приклад 3.22}
{Створити файл з інформацією про 50 потенційних платників
 прибуткового податку за умови, що допускається тільки
 одержання основної заробітної плати і не більш однієї
 заробітної плати за сумісництвом}
type
  t_rec=record
          surname:string;                              {Прізвище}
          MainSalary:LongInt;          {Основна заробітна плата}
          case yes_no:Boolean of          {Є сумісництво чи ні?}
            False:();                        {Немає сумісництва}
            True:(AdditSalary:LongInt;    {Зарплата за сумісн.}
                  PlaceOfWork:string);           {Місце роботи}
  end;
  t_file=file of t_rec;
var
  f:t_file;
  v:t_rec;
  i:Integer;
  c:Char;
begin
  Assign(f,'profit');
  Rewrite(f);
  for i:=1 to 50 do begin
   with v do begin                              {Формування запису}
    WriteLn('Прізвище');ReadLn(surname);
    WriteLn('Основна заробітна плата');ReadLn(MainSalary);
    WriteLn('Є додатковий доход?  Y/N');ReadLn(c);
    if (c='y')or(c='Y') then begin
      yes_no:=True;
      WriteLn('Розмір заробітної плати за сумісництвом');
      ReadLn(AdditSalary);
      WriteLn('Місце сумісництва');ReadLn(PlaceOfWork)
    end
    else yes_no:=False
```

```
    end;
    Write(f,v);                          {Запис компонента у файл}
  end;
  Close(f);
end.
```

3.5.5. Нетипізовані файли

У TP, крім розглянутих, існують також нетипізовані файли, що сумісні з усіма типами файлів і використовуються тоді, коли тип елементів файла не важливий (наприклад, при копіюванні). Ці файли мають такий опис:

var ім'я_файла**: file;**

Наприклад, можливий такий опис:

```
var FileOneType:file;
```

Файл без типу розглядається як послідовність компонентів довільного типу, але обговореного розміру: у нього можна записати значення будь-якої змінної, яка має заданий розмір, а при читанні з такого файла допускається довільна інтерпретація вмісту чергового компонента.

Відкриваються файли без типу тими ж операторами **Reset** і **Rewrite**, але в цьому випадку є другий параметр – розмір запису (компонента файла), заданий у байтах. Попередньо потрібно за допомогою оператора **Assign** зв'язати внутрішнє ім'я файла із зовнішнім:

```
Assign(FileOneType,'f.dat'); Reset(FileOneType,1);
```

Другий параметр операторів **Reset** і **Rewrite** може бути опущений, що означає задання розміру запису в 128 байт. Найбільша швидкість обміну даними забезпечується при довжині запису, кратній 512 байт (розміру сектора на диску).

Потрібно пам'ятати, що якщо загальний розмір файла не кратний обраному розміру запису, то останній запис виявиться неповним, і файл може бути прочитаний не до кінця. Цього не буде при заданні розміру запису рівним 1 байту.

Обмін даними при роботі з нетипізованими файлами здійснюється за участю робочого буфера, за який використовується оголошена в програмі змінна. Її розмір повинен бути достатнім для розміщення даних, що читаються (записуються) за один сеанс читання (запису).

Перед читанням нетипізований файл повинен бути відкритий за допомогою **Reset**, а саме читання здійснюється процедурою

BlockRead(ім'я_файла**,** змінна_буфер**,** кількість_записів**).**

Третій параметр — це кількість записів, які читаються за один раз (тип **Word**).

При виконанні процедури **BlockRead** дані поміщаються в оперативну пам'ять, починаючи з першого байта змінної, зазначеної як другий параметр процедури **BlockRead**. Тому «змінна_буфер» повинна мати розмір, який дорівнює добутку кількості записів, що читаються за один раз (третій параметр), і розміру запису, заданого в процедурі **Reset**. У процедурі **BlockRead** можливе завдання четвертого параметра (тип **Word**), в який поміщається число фактично прочитаних записів.

Запис даних у нетипізований файл проводиться тільки після його відкриття за допомогою **Rewrite**. Для запису даних використовується процедура **BlockWrite**, яка має ті ж три (чи чотири) параметри, що і **BlockRead**. При цьому в «змінну_буфер» потрібно попередньо помістити записи, кількість яких повинна збігатися зі значенням третього параметра процедури **BlockWrite**, а розмір – з другим параметром процедури **Rewrite**. У четвертому параметрі процедури **BlockWrite** (якщо він є) повертається кількість фактично поміщених у файл записів. Якщо на диску немає вільного місця, то після виконання процедури **BlockWrite** значення третього і четвертого параметрів будуть відрізнятися.

Якщо при читанні з диска виявиться, що розмір буфера буде менше зазначеного вище чи при записі на диск недостатньо вільного місця, то при відсутності четвертого параметра в процедурах **BlockRead** і **BlockWrite** буде зафіксована помилка. При наявності четвертого параметра помилка не фіксується.

```
{Приклад 3.23}
{Скопіювати файл. Імена файла-оригіналу і файла-копії
 задаються в діалозі}
const
   CountRead=512;          {Кількість записів, що читаються...}
             {...При довжині записів у 1 байт - розмір буфера}
var
   FileIn,FileOut:file;              {Файл-оригінал, файл-копія}
   NumRead,                         {Число фактично зчитаних...}
   NumWritten:Word;          { ...і фактично записаних записів}
```

```
   Buf:array[1..CountRead] of Char;           {Змінна-буфер}
   Name1,Name2:string; {Імена файла-оригіналу і файла-копії}
begin
   Write('Введи ім''я файла-оригіналу ');
   ReadLn(Name1);
   Write('Введи ім''я файла-копії ');
   ReadLn(Name2);
   Assign(FileIn,Name1);
   {$I-}
   Reset(FileIn,1);                  {Довжина запису 1 байт}
   {$I+}
   if IOResult<>0 then begin
      WriteLn('Файл-оригінал не знайдено');Halt
   end;
   Assign(FileOut,Name2);
   Rewrite(FileOut,1);               {Довжина запису 1 байт}
   repeat
     BlockRead(FileIn,Buf,CountRead,NumRead);
     BlockWrite(FileOut,Buf,NumRead,NumWritten)
   until (NumRead=0)or(NumWritten<>NumRead);
   WriteLn('Копіювання завершене');
   if NumRead<>NumWritten then
     WriteLn('Не вистачило місця на диску для файла ',Name2);
   Close(FileOut);Close(FileIn)
end.
```

У процесі читання/запису за допомогою **BlockRead** і **BlockWrite** поточний покажчик переміщується на ту кількість записів, що було оброблено. Можливе також використання функцій **FileSize** і **FilePos** і процедури **Truncate**, що у даному випадку оперують записами оголошеного в **Reset** чи **Rewrite** розміру.

3.5.6. Спеціальні операції для роботи з файлами

У TP існує кілька процедур для роботи з файловою структурою MS DOS.

Процедура **Rename** служить для перейменування файла чи каталога. Її формат такий:

Rename(файлова_змінна**,** нове_ім'я**).**

Другий параметр задається рядковим виразом і вказує нове зовнішнє ім'я файла чи каталога.

Для знищення файла в TP використовується процедура **Erase**, єдиним параметром якої є внутрішнє ім'я файла:

Erase(файлова_змінна**).**

Ці дві процедури працюють тільки із закритим файлом, але попередньо за допомогою оператора **Assign** файлова змінна (тип якої непринциповий) повинна бути зв'язана з зовнішнім ім'ям файла (чи каталога, якщо перейменовується каталог).

Чотири процедури (**ChDir**, **MkDir**, **RmDir** і **GetDir**) у TP забезпечують роботу з каталогами.

Перші три процедури мають один і той же формат:

ChDir(каталог**), MkDir(**каталог**), RmDir(**каталог**).**

В усіх трьох випадках параметр задається рядковим виразом і вказує ім'я каталога в інтерпретації MS DOS.

Процедура **ChDir** змінює поточний каталог на вказаний, процедура **MkDir** створює новий каталог з вказаним ім'ям, процедура **RmDir** знищує каталог за умови, що він є порожнім.

Процедура **GetDir** дозволяє визначити ім'я поточного каталога на вказаному диску. Формат процедури:

GetDir(диск**,** каталог**),**

де «диск» – вираз типу **Word**, який задає номер диска (0 – активний диск, 1 – диск A, 2 – диск B і т.д.); «каталог» – змінна типу **string**, призначена для повернення шляху до поточного каталога на диску, номер якого вказаний як перший параметр процедури.

```
{Приклад 3.24}
{Застосування спеціальних процедур для роботи з файлами.
 Передбачається, що в поточному каталозі міститься
 файл чи підкаталог з ім'ям tt}
var
   f:file;       {Можна використовувати будь-який файловий тип}
   s:string;
begin
   Assign(f,'tt');
   Rename(f,'ttt');          {Перейменування файла чи каталога}
   MkDir('tmp'); {У поточн. каталозі створений підкаталог TMP}
   ChDir('tmp');                       {Перехід у підкаталог TMP}
   GetDir(0,s);      {Запис у s повного імені поточн. каталога}
           {Якщо поточний каталог C:\USER, то s='C:\USER\TMP'}
   Assign(f,'f.tmp');
   Rewrite(f);                              {Створення файла f.tmp}
          {Продовження програми для обробки...  }
          {...файла f.tmp – потрібна постановка задачі}
   Close(f);                 {Закриття файла для його видалення}
   Erase(f);                              {Видалення файла f.tmp}
   ChDir('..');                            {Перехід у надкаталог}
end.
```

4. ПОКАЖЧИКИ І ДИНАМІЧНІ ДАНІ

4.1. Посилальні типи і покажчики. Тип Pointer

Оскільки пам'ять комп'ютера розбита на сегменти розміром 64 Кбайт (1К=1024 байт), поряд із суцільною нумерацією комірок пам'яті прийнята нумерація усередині сегмента: будь-яка адреса представляється у вигляді двох чисел – номера сегмента і зсуву комірки пам'яті стосовно початку сегмента, у якому вона знаходиться (адреса усередині сегмента). Ці два числа звичайно задаються в шістнадцятковому вигляді і зв'язують двокрапкою (наприклад, $0040:$0048, де $0040 – номер сегмента, $0048 – зсув). Для визначення адреси, за якою розташовується яка-небудь змінна, застосовується операція взяття адреси **@** (наприклад, @i), яка повертає адресу елемента даних у вигляді двох слів: старше – номер сегмента, молодше – зсув. Операція **@** застосовна до будь-яких типів даних.

Для роботи з адресами в TP введений спеціальний посилальний тип, що описується символом « ^ » з наступним ім'ям типу, на значення якого здійснюється посилання:

```
type    p=^Integer;
```

Опис змінних посилального типу провадиться стандартно:

```
type
   pt1=^Integer;
   pt2=record
         Day:Byte;
         Month:string;
         Year:Word
   end;
var
   p1,p2:pt1;
   DatePtr:^pt2;
```

Посилальний тип може бути заданий навіть у випадку, коли базовий тип ще не введений (єдиний виняток в описі). Однак тоді базовий тип повинен бути описаний у тій же послідовності оголошення типів:

```
type
   PtrType=^BaseType;
   BaseType=record
               x,y:Real
   end;
```

Значенням посилального типу є покажчик, тобто адреса першого байта змінної того типу, що є базовим. Присвоєння змінній посилального типу значення здійснюється з використанням операції взяття покажчика (адреси) **@**. Для забезпечення доступу до вмісту адреси, на яку посилається змінна посилального типу, потрібно після імені такої змінної поставити символ « ^ ». Так, у наведеній нижче програмі посилання вигляду p^ є еквівалентним використанню змінної i:

```
type
   t=^Integer;
var
   p:t;
   i:Integer;
begin
   p:=@i;  {В p - адреса i; p^ - те ж, що i}
   ReadLn(i);          {Те ж, що ReadLn(p^) }
   p^:=p^+1;                  {Аналог  i:=i+1}
end.
```

Над змінними посилального типу можна виконувати тільки дві операції порівняння (= та <>) і операцію взяття адреси (**@**). У TP є константа посилального типу **nil**, яка задає покажчик, що є порожнім. Вона сумісна з усіма посилальними типами.

Незважаючи на те, що всі посилальні типи фізично однакові (адреси пам’яті), для різних базових типів вони вважаються різними. Наприклад, помилковим є оператор присвоювання в такій досить простій програмі:

```
type
   t1=^Integer;
   t2=^Real;
var
   p1:t1;
   p2:t2;
begin
   p1:=p2;
end.
```

При роботі з покажчиками можна користуватися явним перетворенням типу. Так, в останньому прикладі оператор p1:=p2 повинен бути записаний у вигляді t2(p1):=p2. Однак при цьому ніяка перевірка на коректність звертань до пам’яті не здійснюється. Наприклад, компілятор не знайде помилку в такій програмі:

```
type
   t1=^Byte;
   t2=^Real;
var
   b:Byte;
   p1:t1;
begin
   p1:=@b;
   t2(p1)^:=-0.5;
end.
```

Тут покажчик t2(p1)^ посилається на область з 6 байт, початок якої збігається з адресою змінної b, що займає 1 байт. У ці 6 байт записане значення –0.5, у результаті чого псується вміст двох описаних підряд змінних b і p1.

Крім типізованих покажчиків, у TP є спеціальний тип **Pointer**, значення якого називають нетипізованим покажчиком. Він сумісний з усіма посилальними типами:

```
type
   t1=^Integer;
   t2=^Real;
var
   p1:t1;
   p2:t2;
   p3:Pointer;
begin
   p3:=p1; p3:=p2; p1:=p3; p2:=p3;  {Припустимо}
   p1:=p2;                          {Неприпустимо}
end.
```

Відзначимо також, що результат операції **@** має тип **Pointer**.

Єдиним можливим значенням типізованої константи-покажчика є **nil**:

```
const  PString:^string=nil;
```

У наведеному нижче прикладі показана можливість використання явного перетворення типу, а також операції взяття адреси для узгодження покажчиків різних типів.

```
{Приклад 4.1}
{Узгодження покажчиків різних типів}
type
  t1=^Byte;
  t2=^string;
```

```
var
  p1:t1;
  p2:t2;
  s1,s2:string;
begin
  p2:=@s1;                                    {p2 посилається на s}
  p2^:='pascal';                                        {Запис у s1}
  p1:=t1(p2);                                {p1 посилається на s1[0]}
  p1^:=3;   {Змінюємо вміст s1[0]; тепер довжина s дорівнює 3}
  WriteLn(p2^);
  s2:='Pascal';
  p1:=@s2;          {Заносимо в p1 адресу першого байта s2,...}
  p1^:=3;     {... тобто посилання на s2[0], і змінюємо s2[0]}
  WriteLn(s2);
end.
```

У програмі в змінну p1 (посилання на тип **Byte**) заноситься в першому випадку посилання на рядок, на який вказує змінна p2, а в другому – на рядкову змінну s2. Далі за рахунок зміни вмісту комірки, на яку посилається p1, в обох випадках здійснюється зміна довжини рядка (p1 посилається на початок рядка, де записана його довжина). В результаті перший оператор друку виводить рядок pas, а другий – Pas.

Змінні посилального типу можуть використовуватися як параметри процедур **Inc** і **Dec**. При цьому зміна значення такої змінної на 1 відповідає зсуву покажчика в пам'яті на кількість байт, що відповідає розміру базового типу. Наприклад, якщо в програмі є описи вигляду

```
var
   m,n,k:LongInt;
   Pt:^LongInt;
```

то в результаті виконання послідовності операторів

```
Pt:=@k;Dec(Pt,2);
```

покажчик Pt буде посилатися на змінну m (Pt=@m), оскільки оператор Dec(Pt,2) «змістить» покажчик Pt на 2*SizeOf(LongInt) байт.

```
{Приклад 4.2}
{Даний масив з n дійсних чисел (n<=100).
 Видалити в ньому всі максимальні елементи}
var
  n,i:Integer;
  a:array[1..100] of Real;
  max:Real;
  Pt:^Real;
begin
  Write('n=');ReadLn(n);
```

```
  for i:=1 to n do begin
    Write('a[',i,']=');ReadLn(a[i]);
  end;
  max:=a[1];
  for i:=2 to n do
    if a[i]>max then max:=a[i];
  Pt:=@a[1];    {У покажчик записуємо адресу першого елемента}
  for i:=1 to n do
    if a[i]<>max then begin
     Pt^:=a[i];{Запис в область пам'яті за допомог. покажчика}
     Inc(Pt);     {Зсув на 6 байт, тобто на наступний елемент}
    end
    else Dec(n);
  WriteLn('У масиві залишилося ',n,' елементів');
  for i:=1 to n do Write(a[i]:8:3);
  ReadLn;
end.
```

У програмі покажчик Pt спочатку посилається на a[1]. Надалі за допомогою процедури **Inc** забезпечується послідовне збільшення адреси, на яку посилається покажчик, на SizeOf(Real) байт, що забезпечує послідовне переміщення його по елементах масиву.

Оскільки змінна типу **PChar** є покажчиком на тип **Char**, значення такої змінної можна змінювати за допомогою процедур **Inc** і **Dec**. Крім того, до даних типу **PChar** можна застосовувати адресну арифметику, а саме, до таких змінних можна додавати ціле значення чи віднімати його. Визначена також операція віднімання для двох операндів, які мають тип **PChar**.

Якщо до елемента даних типу **PChar** додати ціле значення n, то в результаті буде отриманий покажчик типу **PChar**, зміщений у пам'яті на *n* байт. Аналогічно визначається віднімання цілого значення з змінної типу **PChar**.

Різниця двох елементів типу **PChar** визначається як ціле число, що дорівнює кількості байт, на яке зміщений один покажчик стосовно іншого.

До символьних масивів **array of Char** з нульовою базою також можна застосовувати адресну арифметику, однак ім'я такого масиву не може використовуватися як параметр процедур **Inc** і **Dec**, оскільки воно є покажчиком-константою.

```
{Приклад 4.3}
{Випадковим чином сформувати рядок, який не вміщує керуючих
 символів і завершується символом #0. Довжина рядка не повинна
 перевищувати половини області пам'яті, що відводиться під
```

```
 нього. Якщо в цьому рядку кількість символів з непарними
 кодами менше кількості символів з парними кодами, доповнити
 його справа "дзеркальним" відображенням; у протилежному
 випадку залишити рядок без зміни}
var
   x:array[0..100] of Char; {Символьний масив з нульовою базою}
   PCh1,PCh2:PChar;
   n,m1,m2:Integer;
begin
  Randomize; n:=Random(SizeOf(x) div 2)+1;  {n – довжина рядка}
  x[n]:=#0;{Наприкінці рядка розміщуємо завершальний символ #0}
  PCh1:=@x[n];   {Запам'ятовуємо позицію завершального символу}
  for n:=n-1 downto 0 do           {Тут працюємо з масивом x...}
  x[n]:=Chr(Random(223)+32);        {...як зі звичайним масивом}
  WriteLn(x);      {Символьні масиви з нульовою базою можна...}
                  {...виводити на екран і вводити з клавіатури}
  m1:=0;m2:=0;n:=0;
  while (x+n)^<>#0 do begin    {До симв. масивів застосовна...}
    if Odd(Ord((x+n)^)) then Inc(m1);   {...адресна арифметика}
          {Ім'я x - це покажчик на початок масиву, x+n - це...}
         {...покажчик на елемент, зміщений на n позицій від...}
          {...початку масиву, (x+n)^ - значення цього елемента}
    else Inc(m2);
    Inc(n);
  end;
  if m1<m2 then begin
     PCh2:=PCh1-1;  {Застос. адр. арифметики. Покажчик PCh2...}
                {...вказує на попередній байт по відношенню...}
                           {...до символу, на який вказує PCh1}
     while PCh2>=x do begin       {Покажчики можна порівнювати}
       PCh1^:=PCh2^;
       Inc(PCh1);     {До всіх типізованих покажчиків можна...}
       Dec(PCh2)          {...застосовувати процедури Inc і Dec}
     end;
     PCh1^:=#0;               {Дописуємо завершальний символ #0}
  end;
  n:=PCh1-x;         {Різниця - кількість байт між PCh1 і @x[0]}
  WriteLn('Довжина результуючого рядка дорівнює ',n);
  WriteLn(x);
end.
```

У програмі показана можливість застосування адресної арифметики при роботі з символьними масивами і даними типу **PChar**: конструкція x+n служить для організації переходу по елементах символьного масиву, конструкція PCh1–1 служить для одержання покажчика на попередній байт по відношенню до PCh1, а конструкція PCh1–x служить для визначення кількості елементів у символьному масиві.

4.2. Динамічний розподіл пам’яті

4.2.1. Поняття динамічних змінних

Дотепер ми розглядали випадок, коли пам’ять під змінні виділяється статично. Подібні змінні характеризуються тим, що вони мають один і той же формат і розмір на протязі усього часу виконання програми. Під час компіляції для кожного статичного елемента даних виділяється постійний обсяг пам’яті, який під час виконання програми не може бути змінений. Альтернативою статичному є динамічний розподіл, коли пам’ять під змінні виділяється при виконанні програми в міру необхідності. При цьому ускладнюється звертання до подібних змінних, що спричиняє зменшення швидкості обчислень. Для цих цілей у ТР у пам’яті комп’ютера слідом за об’єктним кодом програми виділяється область пам’яті, що розподіляється динамічно, яку дуже часто називають «купою» (heap). Робота з нею здійснюється за допомогою покажчиків.

Ідея, що лежить у концепції покажчиків, полягає в тому, щоб зв’язати певний тип даних з конкретним покажчиком. Сам покажчик у свою чергу теж є елементом даних і являє собою посилання на певну комірку пам’яті. Оскільки покажчик може посилатися на будь-яку адресу пам’яті, потрібно здійснити посилання на адреси в динамічній області пам’яті. У цьому випадку забезпечується оперування вмістом адрес аналогічно тому, як це робиться при оперуванні звичайними змінними, причому ці змінні в програмі не оголошуються, а створюються і знищуються в процесі виконання програми. Такі змінні називаються динамічними.

Розглянемо, наприклад, таку послідовність описів:

```
type
   TDateRec=record
                Day:1..31;
                Month:string[8];
                Year:Integer
   end;
   TDatePtr=^TDateRec;
var
   DateVar:TDatePtr;
```

Другий опис типу констатує, що будь-який об’єкт типу TDatePtr є покажчиком, який буде вказувати на змінну типу TDateRec. Ніякі змінні типу TDateRec можуть не оголошуватися: вони будуть спеціальним чином створюватися в купі в процесі виконання програми. Якщо змінна вже створена, то конструкція DateVar^ (ім’я змінної покажчика із символом « ^ ») розглядається як своєрідне ім’я цієї змінної, тобто DateVar^ — це посилання на динамічну змінну типу TDateRec. Посилаючись за допомогою покажчика на динамічні змінні, можна оперувати ними, не знаючи при цьому, де вони розташовуються.

4.2.2. Процедури і функції, що використовуються при роботі з динамічними даними

Для роботи з динамічними змінними в TP введені процедури **New**, **Dispose**, **GetMem**, **FreeMem**, **Mark** і **Release**. Крім того, перед розміщенням у пам’яті динамічних даних дуже часто використовують функції **MaxAvail**, **MemAvail** і **SizeOf**.

Процедура

New(ім’я_покажчика**)**

розміщує в пам’яті динамічну змінну з ім’ям, яке задається параметром процедури (наприклад, оператор New(DateVar) створює в динамічній пам’яті нову динамічну змінну з адресою, записаною у змінній DateVar; для звертання до цієї змінної необхідно вказати DateVar^). **New** можна викликати і як функцію, яка повертає покажчик того типу, що заданий як її параметр. Наприклад, замість оператора New(DateVar) у програмах можна використовувати оператор DateVar:=New(TDatePtr).

Протилежною процедурі **New** є процедура

Dispose(ім’я_покажчика**),**

що звільняє ділянку пам’яті, на яку посилається покажчик, заданий у вигляді параметра (наприклад, оператор Dispose(DateVar) звільняє пам’ять, що займала змінна DateVar^). При цьому звільнена ділянка пам’яті повертається в системну область, називану пулом вільної пам’яті. В результаті застосування процедури **Dispose** в пам’яті з’являються вільні ділянки (кластери), що не завжди можуть бути надалі використані.

```
{Приклад 4.4}
{У файлі f.dat зберігається 18000 дійсних чисел, що є
 елементами масиву з 10 рядків і 1800 стовпців. Переписати
 в інший файл рядок з максимальною сумою елементів}
{Весь масив не можна розмістити в пам'яті; тому будемо
 створювати 10 рядків, кожний з 1800 чисел}
type
   t1=array[1..1800] of Real;
   t2=array[1..10] of ^t1;
var
   a:t2;                        {Масив з 10 покажчиків на масиви}
   i,j,n:Integer;
   max,s:Real;
   f:file of Real;
   l:LongInt;
begin
   Assign(f,'f.dat');
   Reset(f);
   for i:=1 to 10 do begin
      New(a[i]);                              {Створення і-го рядка}
      s:=0;
      for j:=1 to 1800 do begin
         Read(f,a[i]^[j]);     {Читаємо в елемент динам. масиву}
         s:=s+a[i]^[j];
      end;
      if (i=1)or(s>max )then n:=i;
   end;
   Close(f);
   Assign(f,'f1.dat');
   Rewrite(f);   {Нижче виводимо елементами зміст динамічного}
   for j:=1 to 1800 do Write(f,a[n]^[j]);    {...масиву a[n]^}
   Close(f);
   for i:=1 to 10 do
      Dispose(a[i]);                          {Видалення рядками }
end.
```

Процедури **New** і **Dispose** виділяють і звільняють пам'ять відповідно до того, який обсяг пам'яті визначає створюваний тип (наприклад, у програмі кожна зі змінних a[i]^ займає по 1800×6=10800 байт). Трохи інакше працюють процедури **GetMem** і **FreeMem**.

Процедура

GetMem(ім'я_покажчика**,** кількість_байт**)**

забезпечує виділення пам'яті під динамічну змінну обсягом, який задається другим параметром. Процедура ж

FreeMem(ім'я_покажчика**,** кількість_байт**)**

звільняє задану кількість байт динамічної пам’яті, починаючи з адреси, на яку вказує змінна, що задана як перший параметр.

Необхідно дотримуватися правила, яке полягає в тому, що змінні, розподілені за допомогою **New**, повинні знищуватися процедурою **Dispose**, а пам’ять, відведена за допомогою **GetMem**, повинна звільнятися процедурою **FreeMem** з тим же значенням другого параметра, що й у відповідному звертанні до процедури **GetMem** (змінні, створені за допомогою **New**, можуть знищуватися і процедурою **FreeMem**).

Треба пам’ятати, що ні **New**, ні **GetMem** не можуть відвести під один елемент даних (наприклад, під масив) більше 64K (це обмеження TP, яке розповсюджується і на статичну, і на динамічну пам’ять).

Якщо для динамічної змінної не вистачає пам’яті, то при виконанні процедур **New** і **GetMem** виникає помилка. Тому перед відведенням пам’яті рекомендується здійснювати перевірку наявності достатнього об’єму пам’яті. Мова, насамперед, йде про безперервну область пам’яті, бо під змінну область пам’яті виділяється безперервним фрагментом. Функція без параметрів

MaxAvail

повертає об’єм максимального за розміром фрагмента вільної пам’яті в динамічній області. Значення, що повертається функцією **MaxAvail**, повинне порівнюватися зі значенням, що дорівнює об’єму пам’яті, який необхідно розподілити. Взнати необхідний об’єм пам’яті можна, звернувшись до функції

SizeOf(ім’я_типу**).**

Параметром функції **SizeOf**, як це відзначалося в п.1.6.4, може бути не тільки ім’я типу, але й ім’я змінної.

Функція без параметрів

MemAvail

повертає сумарний об’єм вільної динамічної пам’яті (у байтах). При безладному розподілі та знищенні динамічних змінних динамічна пам’ять виявляється фрагментованою. Тому досить часто функція **MemAvail** повертає значення, яке більше того, котре повертає функція **MaxAvail**.

У наступній програмі показане використання функцій **SizeOf** і **MaxAvail** при роботі з динамічною пам’яттю:

```
type
   tp=record
        x,y:Real;
   end;
var
   point:^tp;
begin
   if MaxAvail>=SizeOf(tp) then New(point)
   else begin
     WriteLn('Немає вільної пам''яті під point');Exit
   end;
   {Продовження програми}
end.
```

Треба, по можливості, знищувати динамічні змінні відразу ж, як тільки відпадає необхідність у них, інакше дуже швидко може вичерпатися вільна пам'ять.

```
{Приклад 4.5}
{У файлі f.dat зберігається більше 10000 дійсних чисел.
 Переписати в інший файл ті з перших 10000 компонентів
 файла f.dat, які перевищують їх середнє арифметичне.
 Якщо 10000 чисел не вміщуються в динамічну пам'ять
 обробити максимальну кількість чисел, що можуть
 бути спільно розміщені в купі}
type t=array[1..10000] of Real;
var
   p:^t;                                    {Покажчик на масив}
   n,                                {Розмір створюваного масиву}
   i:Integer;
   Size:LongInt;{Максимальна розмірність масиву дійсних чисел}
   s:Real;
   f:file of Real;
begin
   n:=10000;        {Орієнтуємося на значення в розділі type }
   Size:=MaxAvail div SizeOf(Real);
   if n<=Size then New(p) {Створення масиву з 10000 елементів}
   else begin          {10000 елементів не розміщаються в купі}
      n:=Size;                    {Змінюємо розмірність масиву}
      GetMem(p,n*SizeOf(Real)) {Створюємо масив з n елементів}
   end;
   Assign(f,'f.dat');Reset(f);
   for i:=1 to n do begin
      Read(f,p^[i]); {Читаємо і-й елемент динамічн. масиву p^}
      s:=s+p^[i];
   end;
   Close(f);
   s:=s/n;
```

```
   Assign(f,'res.dat');Rewrite(f);
   for i:=1 to n do if p^[i]>s then Write(f,p^[i]);
   FreeMem(p,n*SizeOf(Real));      {Знищуємо динамічний масив}
   Close(f)
end.
```

За допомогою процедури

Mark(ім'я_покажчика**)**

можна запам'ятати початок вільної в даний момент області динамічної пам'яті. Якщо надалі звернутися до процедури

Release(ім'я_покажчика**),**

то відбудеться звільнення динамічної пам'яті від останньої зайнятої комірки до адреси, на яку вказує параметр процедури **Release**. Параметр у цих двох процедурах має тип **Pointer**.

4.3. Установка розміру динамічної пам'яті

Динамічні змінні розташовуються в пам'яті в заздалегідь невідомих місцях, причому невідомо, скільки динамічних змінних одночасно буде існувати в програмі, а отже, який об'єм пам'яті потрібний під них. За замовчуванням під купу виділяється весь обсяг наявної оперативної пам'яті, що доступна операційній системі, за винятком пам'яті для коду програми і статичних змінних. Однак деякі програми взагалі не використовують динамічні змінні. Крім того, якщо програма займає всю пам'ять, то вона не в змозі викликати яку-небудь іншу програму.

Для керування розміром динамічної пам'яті служить директива **$M**, яка розміщується на початку програми і має три параметри: максимальний розмір пам'яті, що відводиться під стек локальних змінних підпрограм, мінімальний і максимальний розмір динамічної пам'яті.

Якщо розмір вільної наявної динамічної пам'яті менше значення, що задається другим параметром директиви **$M**, програма не запуститься на виконання. Якщо цей параметр дорівнює 0, то це означає, що програма не вимагає динамічної пам'яті і буде виконуватися завжди.

Максимальний розмір купи (третій параметр) вказує, що розмір купи не буде перевищувати заданого значення. Якщо ж у процесі виконання програма зажадає розмір купи більше зазначеного, то буде помилка.

За замовчуванням провадиться установка **{$M 16384,0,655360}**.

Призначення першого параметра директиви **$M** розглядалося в розділі, присвяченому використанню підпрограм у мові ТР.

4.4. Зв'язані списки

Покажчики є ефективним засобом для побудови списків, тобто таких упорядкованих структур, кожен елемент якої містить посилання, що зв'язує його з попереднім чи наступним елементом. Для організації списків використовуються записи, які складаються з двох змістових частин – основної і додаткової. Основна частина містить інформацію, що підлягає обробці, а в додатковій частині знаходиться покажчик на інший елемент списку. Покажчик на кінець списку зберігається в змінній, котра завжди є присутньою у програмі обробки списків. Якщо в списку немає елементів, тобто він порожній, значення цієї змінної повинне дорівнювати **nil**. У противному разі його перший (останній) елемент містить у додатковій частині **nil**. Основними операціями над списками є формування списку, додавання, вставка, видалення і перегляд елементів.

Найбільш простими є два види списків: стек і черга.

Стек — це список з однією точкою доступу до його елементів, називаною вершиною стека. Додати чи видалити елемент можна тільки через вершину стека. Принцип роботи стека: «останній прийшов – перший вийшов» (LIFO: *Last in – first out*).

Черга — це структура даних, в один кінець якої додаються елементи, а з іншого вилучаються. Принцип роботи черги: «перший прийшов – перший вийшов» (FIFO: *First in – first out*).

Щоб забезпечити можливість роботи зі списком, потрібно включити покажчик до складу запису, що можна зробити, наприклад, так:

```
type
   t=^t1;
   t1=record
         pt:t;
         surname:string[20]
   end;
var
   top,start:t;
```

Тут фігурує виняток із загального правила ТР, яке стверджує, що не можна користуватися яким-небудь ім'ям доти, доки воно не буде

описано: в описі типу t міститься посилання на тип t1, який визначається пізніше.

Тепер немає необхідності створювати масив з покажчиків, оскільки кожний новий запис утримує посилання на попередній.

Для роботи зі стеком, крім покажчика усередині запису, необхідний покажчик для збереження адреси, за якою розташовується вершина стека (таким покажчиком, наприклад, може бути змінна start).

Розглянемо процес створення, обробки і знищення стека на прикладі такої програми:

```
{Приклад 4.6}
{Вводяться прізвища; ознака закінчення введення -
 символ @ як перша літера прізвища. Розташувати дані у стеці.
 Роздрукувати вміст стека без знищення  його. Якщо в
 списку є прізвище "Іванов", вставити в список перед ним
 (у порядку обслуговування) прізвище "Іванова", якщо немає -
 в кінець. Роздрукувати вміст стека зі знищенням елементів}
type
  t=^t1;
  t1=record
        surname:string[20];                    {Основне поле}
        pt:t                                   {Поле-покажчик}
  end;
var
  top,pt1,pt2,v:t;
  s:string[20];
begin
  top:=nil;                                    {Стек порожній}
  pt1:=nil;                        {Попереднього елемента немає}
  ReadLn(s);
  while s[1]<>'@' do begin
     New(top);                        {Створення елемента стека}
     top^.surname:=s;                  {Значення основного поля}
     top^.pt:=pt1;              {Посилання на попередній елемент}
     pt1:=top;                      {Змінюємо попередній елемент}
     ReadLn(s);
  end;
  while pt1<>nil do begin                      {Цикл друку стека}
     WriteLn(pt1^.surname);
     pt1:=pt1^.pt;              {Перехід до попереднього елемента}
  end;
  New(v);                       {Створення елемента, що додається}
  v^.surname:='Іванова';
  if (top=nil)or(top^.surname='Іванов') then begin
        {Якщо стек порожній або перше прізвище - 'Іванов',...}
    v^.pt:=top;     {... вставка у початок стека в порядку...}
```

```
    top:=v                                        {...обслуговування}
  end
  else begin              {Тримаємо два покажчики (pt1 і pt2)...}
    pt1:=top;pt2:=top^.pt;          {...на два сусідніх елементи}
    while not((pt2=nil)or(pt2^.surname='Іванов')) do
    begin
      pt1:=pt2;           {Пошук елемента до відшукання його...}
      pt2:=pt2^.pt                          {...чи вичерпання стека}
    end;
    pt1^.pt:=v;                          {Вставка нового елемента...}
    v^.pt:=pt2                                    {...між pt1 і pt2}
  end;
  pt1:=top;                              {Повернення у вершину стека}
  while top<>nil do begin
     WriteLn(top^.surname);                              {Друк елемента}
     top:=top^.pt;              {Перехід до попереднього елемента}
     Dispose(pt1);                                        {Знищення}
     pt1:=top                            {Суміщення двох покажчиків}
  end;
end.
```

При створенні стека використовуються дві змінні (у програмі top і pt1), одна з яких (top) підтримує посилання на вершину стека, а друга (pt1) – посилання на попередній елемент. Перед створенням стека в них заноситься значення **nil**. На першій ітерації спочатку створюється новий елемент (змінна top^), адреса якого (позначимо її через A1) заноситься в змінну top. Після виконання оператора top^.pt:=pt1 створена динамічна змінна дістає посилання на попередній елемент (на першій ітерації це **nil**, у результаті чого перший елемент стека посилається на **nil**). Наприкінці ітерації (pt1:=top) змінна pt1 також переміщується у вершину стека і буде зберігати адресу A1. На другій ітерації знову створюється елемент top^ (з адресою A2 у змінній top), а оператор top^.pt:=pt1 заносить у її поле pt адресу A1 попереднього елемента, після чого змінна pt1 одержує значення A2. У підсумку два елементи стека виявляються зв'язаними так, що створений новий елемент «знає», де розташовується раніше створений. Перший елемент посилається на **nil**, завершуючи тим самим стек, а на останній елемент (вершину стека) посилаються відразу два покажчики (top та pt1). Процес створення інших елементів нічим не відрізняється від раніше розглянутого процесу.

При обробці даних, які зберігаються в стеці, здійснюється перебір елементів (адрес) до досягнення значення **nil**. Якщо перегляд елементів повинен супроводжуватися їхнім знищенням (як у другому

циклі друку в прикладі 4.6), то при переборі необхідно використовувати два покажчики, один з яких (pt1) використовується власне для переходу від одного елемента до іншого, а другий (top) – для знищення розглянутого елемента після того, як здійснений перехід до попереднього (в порядку створення) елемента.

Процеси створення й обробки черги подібні з аналогічними процесами для стека. Відмінність обумовлена, насамперед, необхідністю підтримки в черзі двох точок входу (покажчиків) – адреси початку (для обробки) і адреси кінця (для поповнення) черги.

```
{Приклад 4.7}
{Вводяться дійсні числа до натискання <Ctrl/Z>. Роздрукувати
 в порядку введення усі введені числа, попередньо виключивши
 мінімальний елемент (якщо їх декілька, то виключити перший)}
type
  t=^t1;
  t1=record
        value:Real;                              {Основне поле}
        pt:t                                     {Поле-покажчик}
  end;
var
  pt1,pt2,start,finish:t;
  min:Real;
begin
  start:=nil;finish:=nil;                        {Черга порожня}
  while not Eof do begin          {Цикл до натискання <Ctrl/Z>}
     New(pt1);                          {Створюємо новий елемент}
     ReadLn(pt1^.value);                {Визначаємо основне поле}
     if start=nil then start:=pt1           {Якщо це перший...}
                  {...елемент, запам'ятовуємо початок черги}
     else finish^.pt:=pt1;      {Інакше попередній елемент...}
                                        {...зв'язуємо з новим}
     pt1^.pt:=nil;           {Останній елемент зв'язуємо з nil}
     finish:=pt1;           {Запам'ятовуємо адресу кінця черги}
  end;
  if start=nil then WriteLn('Черга порожня')
  else                                          {Черга не порожня}
  begin
     min:=start^.value;              {Починаємо шукати мінімум}
     pt1:=start^.pt;    {Переходимо до другого елемента черги}
     while pt1<> nil do begin           {Цикл до кінця черги}
        if pt1^.value<min then min:=pt1^.value;
        pt1:=pt1^.pt          {Перехід до наступного елемента}
     end;
     pt2:=start;                  {Запам'ятовуємо початок черги}
     if start^.value=min then          {Перший – мінімальний?}
```

```
      start:=start^.pt                {Зміщуємо початок черги}
    else                             {Перший не є мінімальним}
    begin
      pt1:=start;                     {Знову на початок черги}
      pt2:=start^.pt;
      while pt2^.value<>min do begin
        pt1:=pt2;pt2:=pt2^.pt
      end;         {pt1 - перед мінімумом, pt2 - на мінімумі}
      pt1^.pt:=pt2^.pt;          {Зв'язуємо елементи перед...}
    end;                       {...мінімумом і після мінімуму}
    Dispose(pt2);     {Видаляємо елемент із мінім. значенням}
    pt1:=start;                       {Знову на початок черги}
    if start=nil then WriteLn('Черга порожня')
    else    {Для видал. потрібні два покажчики: pt1 і start}
    while start<>nil do begin           {Цикл до кінця черги}
       WriteLn(start^.value);
       start:=start^.pt;               {Перехід до наступного}
       Dispose(pt1);                       {Знищення елемента}
       pt1:=start                       {Суміщення покажчиків}
    end;
  end
end.
```

На початку роботи програми змінні start і finish дістають значення **nil**, що свідчить про порожнечу черги. При створенні першого елемента змінна start посилається на нього і більш не змінюється, зберігаючи посилання на початок черги. У першого елемента немає попереднього, тому зв'язок попереднього елемента зі знову створеним здійснюється тільки з моменту створення другого елемента черги. В усіх випадках останній створений елемент посилається на **nil** (pt1^.pt:=nil), а змінна finish одержує посилання на останній створений елемент, зберігаючи тим самим посилання на кінець черги. Для перегляду (обробки) елементів черги (пошук мінімуму) досить однієї змінної (pt1), у той час як обробка черги з її знищенням (у програмі вивід на екран) вимагає використання двох посилальних змінних (у програмі start і pt1).

У програмі показаний принцип видалення елементів з черги: для цього досить зв'язати два елементи – перед тим, що видаляється, і безпосередньо наступний за ним.

Відзначимо, що знищення за допомогою **Dispose** вилучених з черги (чи стека) елементів не є обов'язковим, але воно бажано з-за того, що в противному випадку в динамічній пам'яті залишаються зайняті ділянки пам'яті, на які відсутні посилання.

5. ПІДПРОГРАМИ

5.1. Функції

Нехай потрібно написати програму обчислення значення факторіалу числа 12 (12!=1·2·…·12). Вона може мати такий вигляд:

```
var
  i,factorial:LongInt;
begin
  factorial:=1;
  for i:=1 to 12 do
     factorial:=factorial*i;
  WriteLn('12!=',factorial)
end.
```

Якщо нам знадобиться обчислити не 12!, а 7!, то в програмі усюди потрібно буде здійснити заміну 12 на 7. Це легко зробити в даній програмі, але як бути у випадку написання програми для розв'язання складної задачі? Тут на допомогу можуть прийти константи, адже зміни тоді будуть провадитися тільки в розділі опису констант:

```
const
  n=12;
var
  i,factorial:LongInt;
begin
  factorial:=1;
  for i:=1 to n do
     factorial:=factorial*i;
  WriteLn(n,'!=',factorial)
end.
```

Більш розумно використовувати змінну, що забезпечить незалежність тексту програми від початкових даних:

```
var
  i,n:Integer;
  factorial:LongInt;
begin
  Write('Введи ціле невід''ємне число до 12 ');
  ReadLn(n);  factorial:=1;
  for i:=1 to n do
     factorial:=factorial*i;
  WriteLn(n,'!=',factorial)
end.
```

Але як бути, якщо в якій-небудь складній програмі в різних місцях потрібно буде обчислювати факторіали різних значень? Що ж, повторювати ті самі оператори, що реалізують метод обчислення факторіала багаторазово? Так, можлива і така реалізація, але це недоцільне. Більш доцільно використовувати підпрограму-функцію. Напишемо подібну програму, але таку, що використовує функцію.

```
{Приклад 5.1}
{Дано N об'єктів. Скільки можна сформувати з них різних
 сполучень, що містять M об'єктів. Число сполучень з N за M
 визначається формулою N!/((N-M)!*M!)}
var
  n,m:Integer;
function factorial(k:Integer):LongInt;
var
  i,f:LongInt;
begin
  f:=1;
  for i:=1 to k do f:=f*i;
  factorial:=f
end;
begin
   WriteLn('Введи n i m (n>=m )');
   ReadLn(n,m);
   WriteLn('Число сполучень з ',n,' по ',m,' дорівнює ',
           factorial(n) div (factorial(n-m)*factorial(m)))
end.
```

Як це випливає з наведеної програми, використання функцій дає дві основних переваги:

1) відокремлюється хід виконання головної програми від конкретної реалізації підпрограми: немає потреби знати, як реалізована підпрограма, потрібно тільки вміти до неї звернутися;

2) опис функції в програмі дається один раз, а використовувати її можна багаторазово (у наведеній програмі замість того, щоб тричі реалізовувати метод обчислення факторіала, здійснене триразове звертання до оголошеної на початку програми функції).

Структура функції подібна зі структурою програми (заголовок і тіло функції, що складається з розділу опису і здійсненої частини. Розглянемо особливості опису функції і звертання до неї.

1. Функція оголошується в розділі описів програми.

2. Заголовок функції починається зі службового слова **function**, слідом за яким йде ім'я функції.

3. Після імені в круглих дужках перелічуються формальні параметри і їхні типи (у даному випадку один параметр *k* типу **Integer**). Якщо параметрів декілька, то їхнє оголошення здійснюється через крапку з комою. Якщо кілька параметрів мають один і той же тип, то їх можна перелічити через кому з вказівкою спільного типу; наприклад,

```
function fun(a,b,c:Real; d:Integer):Real;
```

4. Якщо функція не має формальних параметрів, то круглі дужки в заголовку не ставляться.

5. Після дужок, у які укладені формальні параметри, через двокрапку вказують тип функції (тип значення, що повертається). Функція служить для обчислення тільки одного значення, що передається в програму через ім'я функції (хоча, як це буде показано в підрозділі 5.2, через так називані **var**-параметри функція може повернути і кілька результатів, але це поганий стиль програмування на ТР). У зв'язку з цим тип значення, яке повертається, – це будь-який скалярний тип (числовий, символьний, булевський, переліку), рядковий тип чи покажчик.

6. Тип значення, що повертається, може бути заданий тільки ім'ям. Не можна, наприклад, вказати тип string[20]; у цьому випадку програмісту необхідно новий тип (наприклад, type t_st20=string[20]) і використовувати його при вказівці типу результату, що повертає функція.

7. Тип формальних параметрів теж задається ім'ям (можливо також використання типу «відкритий масив», наприклад **array of Real**). При необхідності використання як параметрів масивів, записів і інших даних, типи яких не задаються одним ім'ям, у розділі опису типів програми вводять нові типи даних (наприклад, type t_array=array[1..10] of Real) і використовують їх при описі формальних параметрів (див. приклад 5.2).

8. Деякі з параметрів можуть не мати типу (див. підрозділ 9.5).

9. Усередині функції перед її здійсненою частиною можуть бути оголошені змінні, котрі називаються локальними. Ці змінні поза функцією не існують, оскільки під них виділяється пам'ять щораз при виклику функції, а при виході з функції ці змінні знищуються.

10. Усі змінні, описані в програмі до оголошення функції, також відомі всередині функції. Ці змінні називаються глобальними. Щоб

уникнути помилок, глобальні змінні всередині функції використовувати не рекомендується.

11. Ім'я функції в частині, що виконується, повинно дістати своє значення, інакше результат роботи функції не буде переданий у програму.

12. Функція завершується службовим словом **end**, після якого ставиться крапка з комою.

13. Для звертання до функції необхідно згадати її ім'я в якому-небудь операторі (присвоювання, друку і т.д.). При цьому замість формальних параметрів повинні бути підставлені фактичні параметри, у якості яких можуть виступати змінні, константи, вирази. Типи формальних і фактичних параметрів повинні бути сумісні з присвоювання. Якщо в програмі активізована директива компіляції **$X** (стан **{$X+}** – розширений синтаксис), то до функції можна звернутися як до окремого оператора (наприклад, factorial(10);). Однак це можна робити тільки у випадках, коли значення, що повертається, не використовується. За замовчуванням установлена директива **{$X-}**.

14. Імена локальних змінних не можуть збігатися з іменами формальних параметрів.

15. Імена формальних параметрів і локальних змінних можуть збігатися з різними іменами, оголошеними в розділі описів головної програми. В цьому випадку в тілі функції активні імена, що оголошені в ній.

```
{Приклад 5.2}
{Обчислити слід (суму елементів на головній діагоналі)
 квадратної матриці розміру не більше ніж 20x20}
type
   t20x20=array[1..20,1..20] of Real;  {Вводимо тип для масиву,
                     щоб можна було описати формальний параметр}
function TraceOfMatr(a:t20x20;n:Byte):Real; {Заголовок функції}
var
  i:Byte;
  sum:Real;
begin
  sum:=0;
  for i:=1 to n do sum:=sum+a[i,i];
  TraceOfMatr:=sum        {Результат заноситься в ім'я функції}
end;                                          {Кінець функції}
var
  matrix:t20x20;            {Для забезпечення сумісності типів}
  i,j,n:Byte;
```

```
begin
  Write('Введіть розмірність, число <=20 ');
  ReadLn(n);
  WriteLn('Введіть рядками елементи матриці через <Enter>');
  for i:=1 to n do
    for j:=1 to n do
     ReadLn(matrix[i,j]);
  WriteLn('Сума дорівнює ',TraceOfMatr(matrix,n))      {При ...}
    {...звертанні до функції замість формальних параметрів ...}
end.      {...a та n підставлені фактичні параметри matrix і n}
```

У програмі оголошений тип **t20x20**, оскільки в список формальних параметрів не можна включити опис

```
a:array[1..20,1..20] of Real
```

5.2. Процедури

Як уже говорилося, правила гарного тону в програмуванні на ТР вимагають, щоб функція повертала тільки одне значення. А як бути, якщо підпрограма в процесі своєї роботи або взагалі не повинна повертати яке-небудь значення (наприклад, підпрограма для виводу таблиці), або повинна змінювати значення відразу декількох змінних? У цьому випадку підпрограму оформлюють у вигляді процедури.

Опис процедури подібний з описом функції. По суті, існують тільки такі відмінності:

- у заголовку замість службового слова **function** використовується слово **procedure**;
- у заголовку не вказується тип імені процедури;
- у тілі процедури не здійснюється присвоювання значення імені процедури.

Важливим моментом при використанні процедур є застосування так званих **var**-параметрів (їх можна використовувати й у функціях, але це поганий стиль програмування на ТР). Якщо в процесі роботи необхідно змінити значення декількох змінних чи значення якої-небудь структурованої змінної, то відповідні імена можуть бути не згадані в списку формальних параметрів (тобто відповідати глобальним змінним). Однак можна в списку параметрів перед декількома іменами задати службове слово **var**, яке вказує на те, що дані імена визначають змінні, що змінюють свої значення в процесі роботи підпрограми. У списку формальних параметрів процедури

можуть зустрічатися в будь-якім сполученні параметри-значення і параметри-змінні (**var**-параметри).

Приклад запису заголовка процедури:

```
procedure FindMaxMin(x,y:Integer;var max,min:Integer);
```

У чому ж відмінність між двома типами змінних? Дотепер ми мали справу тільки з передачею параметрів за значенням. Цей метод полягає в тому, що значення фактичного параметра призначається відповідному формальному параметру. Як тільки починається виконання процедури чи функції, ніякі зміни значення формального параметра не впливають на значення відповідного фактичного параметра. Такий спосіб діє за замовчуванням, тобто в усіх випадках, коли не вказаний інший спосіб.

Використання **var**-параметра забезпечує передачу параметра за адресою. У цьому випадку в процедуру (чи функцію) пересилається вже не значення фактичного параметра, а його місце розташування в пам'яті (адреса). Якщо формальний параметр має атрибут **var**, то будь-які зміни формального параметра будуть відбиватися в значеннях фактичного параметра, оскільки вони тепер займають одну й ту ж область пам'яті.

Якщо виклик функції може з'являтися в будь-якому виразі, то цього не можна сказати про процедуру. Щоб звернутися до процедури, досить згадати її ім'я разом зі списком фактичних параметрів як окремого оператора. Як фактичні параметри, що підставляються замість формальних параметрів, оголошених зі службовим словом **var**, можна використовувати тільки змінні (але не константи і вирази).

```
{Приклад 5.3}
{Скласти процедуру, яка видаляє з першого рядка
 усі символи, що входять в другий рядок}
procedure DelSymbols(var st1:string;st2:string);
var
   i:Integer;
begin
   for i:=1 to Length(st2) do begin
     while Pos(st2[i],st1)<>0 do
       Delete(st1,Pos(st2[i],st1),1)
   end
end;
{Кінець процедури}
```

```
var
   s1,s2:string;
begin
   ReadLn(s1);
   ReadLn(s2);
   DelSymbols(s1,s2);                    {Звертання до процедури}
   WriteLn(s1);
end.
```

Відзначимо наступне: усе, що може виконати процедура, є здійсненним і за допомогою функції (оскільки апарат **var**-параметрів використовується й у функціях). Справедливим є і зворотне твердження.

Ще один момент у використанні процедур і функцій складається в можливості застосування випереджального опису, який полягає в тому, що використовувана підпрограма може бути описана тільки завданням свого заголовка, слідом за яким йде стандартне ім'я **forward**. Опис тексту такої підпрограми може бути розташований в будь-якому місці розділу описів без повторення списку формальних параметрів (хоча його можна і повторити). Випереджальний опис необхідний у випадку взаємного звертання процедур чи функцій один до одного:

```
procedure first(x,y:Real);forward;    {Випереджальний опис}
procedure second(x,y:Integer);  {Початок процедури second}
begin
  {...}
  first(-1, 2.7);
  {...}
end;                                  {Кінець процедури second}
procedure  first;                {Опис тексту процедури first}
begin
  {...}
  second(-7, 12);
  {...}
end;                                   {Кінець процедури first}
```

Якщо необхідно вийти з процедури чи функції, не досягши її кінця, то рекомендується використовувати процедуру **Exit**. Перед виконанням процедури **Exit** ім'я функції і **var**-параметри повинні одержати значення.

Локальні змінні, як це відзначалося раніше, створюються в пам'яті в момент активізації підпрограми і знищуються при виході з неї. Якщо програма викликає іншу підпрограму, то локальні змінні викликуваної підпрограми розміщуються в пам'яті слідом за

локальними змінними першої підпрограми, і утвориться ланцюжок груп локальних змінних. Знищення локальних змінних провадиться з хвоста цього ланцюжка. Цей принцип відведення і звільнення пам'яті відповідає дисципліні обслуговування, називаної стеком. Стек не може займати більше 65520 байт. За замовчуванням розмір стека дорівнює 16384 байт. Якщо в програмі не використовуються підпрограми, можна зменшити розмір стека, а якщо вони використовуються активно, – збільшити. Використовуваний програмою розмір стека задається першим параметром директиви компіляції **$M** (наприклад: {$M 5000,0,0}), що вказується на самому початку програми. Меншим, ніж 1024, цей параметр задати не можна.

При переповненні стека можливі помилки. Для забезпечення контролю переповнення стека використовують директиву **$S**: якщо вона має вид **{$S+}**, то компілятор генерує код, що перевіряє стан стека перед відведенням пам'яті під локальні змінні. При завданні директиви **{$S-}** контроль не провадиться.

5.3. Рекурсивні підпрограми

Кожна з підпрограм може викликати інші підпрограми, у тому числі звертатися до самої себе. Підпрограма, що звертається до самої себе, називається *рекурсивною*. Рекурсія може бути і неявною, коли підпрограма first викликає підпрограму second, а та у свою чергу викликає підпрограму first. У зв'язку з наявністю взаємних посилань підпрограм однієї на одну виникає проблема, обумовлена вимогою TP до описів: спочатку описати, потім використовувати. При неявній рекурсії опис підпрограм здійснюється з використанням випереджального опису за допомогою описової директиви **forward** (див. підрозділ 5.2).

Обов'язковим елементом в описі будь-якого рекурсивного процесу є деяке твердження, що визначає умову завершення рекурсії; іноді воно називається опорною умовою. В ній може бути задане яке-небудь фіксоване значення, яке обов'язково буде досягнуте в ході рекурсивного обчислення, дозволяючи тим самим організувати своєчасну зупинку процесу виконання обчислень. Крім того, повинен бути другий елемент – спосіб вираження одного кроку розв'язання за допомогою іншого, більш простого. Кількість рівнів вкладеності може бути великою.

```
{Приклад 5.4}
{Рекурсивна функція обчислення факторіала: 0!=1, k!=k*(k-1)!}
function factorial1(k:Integer):LongInt;
begin
  if k=0 then factorial1:=1      {Якщо опорна умова виконана}
         else factorial1:=k*factorial1(k-1)          {Якщо ні}
end;
var
  n:Integer;
begin
  ReadLn(n);
  WriteLn(factorial1(n))
end.
```

Рекурсивною може бути і процедура.

```
{Приклад 5.5}
{Визначити множення цілих чисел через додавання:
 x*0=0, x*y=x+x(y-1) при y>0}
procedure mult(x,y:LongInt;var z:LongInt);
begin
 if y<0 then begin y:=-y;x:=-x end;
 if y=0 then z:=0                                    {Опорна умова}
        else begin mult(x,y-1,z);z:=x+z end{Рекурсивний виклик}
end;
var
 n1,n2,z:LongInt;
begin
 ReadLn(n1,n2);
 mult(n1,n2,z);
 WriteLn(z)
end.
```

5.4. Відкриті масиви і рядки. Константні параметри

У TP 7.0 введені так називані відкриті агрегатні параметри – масиви і рядки: якщо у якості формального параметру використовується одновимірний масив, то при його описі можна не задавати границі зміни індексу. Сам параметр при цьому повинний бути описаний зі службовим словом **var**, наприклад, у виді

```
var a:array of Real;
```

У якості відповідного фактичного параметра підпрограми в цьому випадку може бути одновимірний масив будь-якого розміру (однак типи елементів повинні збігатися). Нижня межа відкритого масиву завжди дорівнює 0. Тому реальні межі переданого масиву визначають

всередині підпрограми за допомогою функцій **Low** і **High**: функція **Low** дає в цьому випадку завжди значення 0, а **High** – відмінне від 0 верхнє значення індексу при звертанні до масиву – фактичного параметра (кількість елементів масиву мінус 1).

```
{Приклад 5.6}
{Чи правда, що сума парних елементів масиву
 більше суми непарних}
function compare(var a:array of Integer):Boolean;
var
   i:Word;
   SumEv,SumUnev:LongInt;             {SumEv – сума парних,...}
begin                      {...SumUnev – сума непарних елементів}
   SumEv:=0;SumUnev:=0;
   for i:=Low(a) to High(a) do
   begin
     if Odd(a[i]) then SumUnev:=SumUnev+a[i]
                  else SumEv:=SumEv+a[i];
   end;
   compare:=SumEv>SumUnev;
end;
var
   a:array[-1..7] of Integer;
   i:Integer;
begin
   WriteLn('Введіть елементи масиву');
   for i:=-1 to 7 do Read(a[i]);
   WriteLn(compare(a));
end.
```

Відкриті рядки в TP 7.0 описуються типом **openstring** і також повинні супроводжуватися службовим словом **var**. У цьому випадку максимальна довжина формального параметра завжди буде збігатися з максимальною довжиною рядка, що переданий в підпрограму як фактичний параметр. Замість типу **openstring** **var**-параметр може бути описаний з типом **string**, але при цьому на початку програми повинна бути встановлена директива компіляції **{$P+}**. Якщо в програмі встановлена директива **{$P-}**, то **var**-параметри типу **string** мають максимальну довжину, що дорівнює 255. Щоб у цьому випадку можна було замість нього підставити рядок, який описаний з іншою максимальною довжиною, потрібно на початку програми встановити директиву **{$V-}**. При цьому знімається контроль відповідності типів рядкових параметрів при звертанні до підпрограм. За замовчуванням встановлені директиви **{$V+}** і **{$P-}**.

У TP 7.0 введені так називані константні параметри

```
procedure p(const par:Real);
```

Оголошення формального параметра зі службовим словом **const** забороняє зміни відповідного параметра всередині підпрограми, і поява операторів, що модифікують такі параметри, спричиняє діагностику помилки на етапі компіляції.

5.5. Дальня і ближня моделі виклику підпрограм

В описі слідом за заголовком процедури чи функції через крапку з комою може бути вказана одна з директив **far** чи **near**. При завданні директиви **near** створюється більш швидкодіюча підпрограма, але виклик такої підпрограми можливий тільки з того модуля (див. главу 6), в якому вона оголошена. Ця модель виклику називається ближньою і встановлюється за замовчуванням. Якщо вказана директива **far**, то відповідна підпрограма може бути викликана з будь-якого місця програми (забезпечується дальня модель виклику), але при цьому підпрограма виконується значно повільніше. Фрагмент програми може бути обведений директивою компіляції **$F**: спочатку **{$F+}**, потім **{$F-}**. У цьому випадку всі підпрограми, що описані в цьому фрагменту, компілюються з дальньою моделлю виклику **({$F+}** – вмикання дальньої моделі, **{$F-}** – відключення дальньої моделі).

5.6. Процедурний тип

У TP можливе використання змінних, приймаючих значення процедур і функцій. Тип таких змінних задається службовим словом **procedure** чи **function** зі списком параметрів (якщо він передбачається) і типом значення, що повертається, для функції. Ім’я процедури чи функції в описі типу природно не ставиться:

```
type
   proc1=procedure;
   proc2:procedure(a:Integer; var b:Real);
   func:function(x,y:Real):Real;
var
   p1:proc1;
   p2:proc2;
   f:func;
```

Тут p1 – процедура без параметрів, p2 – процедура з параметрами, f – функція.

Якщо тепер у користувальницькій програмі описати процедуру чи функцію зі структурою заголовка, аналогічній опису типу якої-небудь зі змінних p1, p2, f, то цій змінній можна «присвоїти» відповідну процедуру чи функцію (зв'язати змінну з процедурою чи функцією). Після цього можна працювати з такою змінною точно так само, якби вона фігурувала як ім'я відповідної процедури або функції. Так, якщо в програмі описана функція

```
function hypoten(a,b:Real):Real;
begin
   hypoten:=Sqrt(a*a+b*b)
end;
```

то до неї можна звернутися подвійно:

```
WriteLn(hypoten(1.2, 10))
```

або

```
f:=hypoten; WriteLn(f(1.2, 10)).
```

Процедурний тип дозволяє за допомогою однієї змінної виконувати різні процедури (функції), якщо ці процедури (функції) мають однакову структуру заголовка.

Вимоги, які необхідно виконати при використанні процедурного типу:

1. Підпрограма, що присвоюється процедурній змінний, повинна компілюватися в режимі дальнього виклику (з директивою **{$F+}** чи з вказівкою після заголовка процедури або функції службового слова **far**).

2. Процедурній змінний не можна присвоїти стандартну процедуру чи функцію.

3. Такі процедури і функції не можуть бути вкладеними в інші підпрограми.

Змінні процедурного типу можуть оголошуватися як типізовані константи. Для цього в оголошенні після знака рівності потрібно вказати ім'я процедури чи функції (без завдання параметрів), яке є початковим значенням процедурної змінної, що оголошується. Так, наприклад, після того, як будуть описані наведені вище тип tfunc і функція hypoten, можна включити в програму такий опис:

```
const  fn:func=hypoten;
```

```
{Приклад 5.7}
{Написати процедури, які видаляють із заданого рядка всі
 символи, що входять в другий рядок (перша процедура), чи всі
 підрядки, що збігаються з другим рядком (друга процедура)  }
type                                    {Оголошення процедурного...}
  p=procedure(var st1:string;st2:string);              {...типу}
{$F+}                                   {Вмикання дальнього виклику}
procedure del_symb(var st1:string;st2:string);    {Процедура 1}
var
  i,j:Integer;
begin
  for i:=1 to Length(st2) do
  begin
    j:=1;
    while j<=Length(st1) do
      if st1[j]=st2[i] then  Delete(st1,j,1) else j:=j+1
  end
end;
procedure del_substr(var st1:string;st2:string);  {Процедура 2}
begin
  while Pos(st2,st1)<>0 do
    Delete(st1,Pos(st2,st1),Length(st2))
end;
{$F-}                                 {Відключення дальнього виклику}
var
  s1,s2:string;
  v:p;                              {Оголошення процедурної змінної}
  c:Char;
begin
  WriteLn('Введи оброблюваний рядок');ReadLn(s1);
  WriteLn('Введи рядок для порівняння');ReadLn(s2);
  WriteLn('Видаляти символи/рядок? y/n');
  ReadLn(c);
  if c in ['y','Y'] then v:=del_symb {Визначення значення...}
  else v:=del_substr;                    {...процедурної змінної}
  v(s1,s2);                {Використання процедурної змінної...}
                                    {...для виклику підпрограми}
  WriteLn(s1);
end.
```

Процедурний тип дозволяє використовувати процедури і функції як параметри інших процедур і функцій. При цьому як фактичні параметри можуть використовуватися як процедурні змінні, так і безпосередньо імена підпрограм. У цьому випадку обов'язково повинен бути оголошений відповідний процедурний тип у розділі **type**, який і буде використовуватися при описі відповідного формального параметра.

```
{Приклад 5.8}
{Реалізувати процедуру відшукання кореня рівняння f(x)=0
 на відрізку [a1, a2] у припущенні, що на цьому відрізку
 існує єдиний корінь. Використовувати метод розподілу
 відрізка навпіл. Корінь рівняння f(x)=0 відшукувати
 з точністю r>0 : x0 - корінь, якщо довжина розглянутого
 відрізка не перевершує r. Знайти корені рівнянь
 f(x)=Sin(x) і f(x)=(Sin x)(Sin x)-3(Cos x)(Cos x)/4     }
type
  f_type= function (x:Real):Real;
procedure root(var x:Real;a1,a2:Real;eps:Real;f:f_type);
                    {Формальний параметр f має тип "функція"}
begin
  while Abs(a2-a1)>eps do begin
    x:=(a1+a2)/2;
    if f(a1)*f(x)<=0 then a2:=x else a1:=x;
  end;
end;
{$f+}
function f1(x:Real):Real; {Процедурна змінна не може мати...}
             {...стандартне ім'я (тут Sin). Здійснено заміну}
begin
  f1:=Sin(x);
end;
function f2(x:Real):Real;
begin
  f2:=(Sin(x))*(Sin(x))-1.75*(Cos(x))*(Cos(x))
end;
{$f-}
var
  a1,a2,r,x:Real;
  i:Word;
  ff:f_type;
begin
  for i:=1 to 2 do
  begin
    WriteLn('Введіть ліву і праву межі інтервалу');
    ReadLn(a1,a2);          {Інтервал повинен містити корінь!}
    Write('Введіть точність: ');ReadLn(r);
    {Нижче два варіанти фактичних параметрів процедурного
     типу: при i=1 - ім'я функції, при i=2 - змінна
     процедурного типу}
    if i=1 then root(x,a1,a2,r,f1)
    else begin ff:=f2;root( x,a1,a2,r,ff);end;
    WriteLn(x)
  end
end.
```

5.7. Функції, що повертають покажчик

Як це відзначалося в підрозділі 5.1, функції можуть повертати покажчик. Це відкриває можливість для побудови більш гнучких функцій з імітацією повернення значень у вигляді масивів і інших складних типів даних.

Програма, що наводиться нижче, містить функцію, яка, повертаючи значення у вигляді покажчика, фактично забезпечує повернення одновимірного масиву, який є рядком двовимірного числового масиву.

```
{Приклад 5.9}
{Даний двовимірний масив цілих чисел, розміру не більше 10x10.
 Переписати в одновимірний масив рядок, який містить
 найбільшу кількість непарних елементів. Оформити функцію,
 що повертає покажчик на такий рядок двовимірного масиву
 чи nil, якщо в двовимірному масиві немає непарних елементів}
type
  T10=array[1..10] of Integer;   {Тип для одновимірного масиву}
  T10x10=array[1..10] of T10;     {Тип для двовимірного масиву}
  TPt10=^T10;{Посилальний тип- посилання на одновимірний масив}
function f(q:T10x10;m,n:Integer):TPt10;
var
  i,j:Integer;
  max,num:Integer;
begin
  max:=-1;
  for i:=1 to m do  begin
    num:=0;
    for j:=1 to n do
      if Odd(q[i,j]) then Inc(num);
    if num>max then begin
        max:=num;
        f:=@q[i,1];          {Запам'ятовуємо адресу i-го рядка}
    end;
  end;
  if max=0 then f:=nil; {За відсутністю непарних елементів...}
end;                                  {...повертаємо значення nil}
var
  q:T10x10;
  d:T10;
  i,j:Integer;
  m,n:Integer;
begin
  Write('Введи кількість рядків: m=');ReadLn(m);
  Write('Введи кількість стовпців: n=');ReadLn(n);
```

```
  for i:=1 to m do
     for j:=1 to n do begin
        Write('q[',i,',',j,']=');ReadLn(q[i,j]);
     end;
  d:=f(q,m,n)^;      {У масив d переписуємо одновимірний...}
                  {...масив, адресу якого повертає функція f}
  WriteLn('Більше всього непарних елементів у рядку');
  for  j:=1 to n do
     Write(d[j]:8);
  WriteLn;
end.
```

У даній програмі використана особливість розподілу пам'яті, яка полягає в тому, що двовимірні масиви розміщуються в пам'яті безперервною ділянкою рядками, починаючи з першого рядка. Конструкція @q[i,1] позначає адресу першого елемента i-го рядка двовимірного масиву. Оскільки ім'я функції має посилальний тип з базовим типом array[1..10] of Integer, то після виконання оператора присвоювання f:= @q[i,1] в імені функції буде міститися покажчик на i-й рядок масиву q. Оператор d:=f(q,m,n)^ забезпечує переписування у масив d одновимірного масиву, на який вказує повернутий функцією f покажчик. Він також демонструє методику доступу до значення, що зберігається за адресою, яка повертається функцією у вигляді покажчика (символ «^» розташовується після переліку параметрів функції).

Відзначимо також, що при відсутності в масиві непарних елементів, функція поверне порожній покажчик **nil**, що зробить помилковим оператор d:=f(q,m,n)^, у зв'язку з чим, починаючи з цього оператора, в програму треба внести такі зміни:

```
Pt:=f(q,m,n);
if Pt=nil then Write('У масиві немає непарних елементів')
else begin
   d:=Pt^;
   WriteLn('Більше всього непарних елементів у рядку');
   for  j:=1 to n do Write(d[j]:8);
end;
WriteLn;
```

описавши змінну Pt як змінну типу **TPt10**. У цьому випадку в масив d здійснюється переписування даних тільки при поверненні функцією f значення, відмінного від **nil**.

Можливість повернення функцією значення у вигляді покажчика дозволяє організувати в програмі меню, що базується на поверненні

функцією покажчиків на процедурний (функціональний) тип, що демонструє такий приклад.

```
{Приклад 5.10}
{Здійснити табулювання однієї з двох функцій із трьома
 коефіцієнтами (a, b, c) на заданому інтервалі [d1, d2]
 для n рівновіддалених точок. Побудувати підпрограму
 табулювання, яка залежить від функції з чотирма параметрами
 (3 коефіцієнти і значення аргументу). Для встановлення
 функції, що табулюватиметься, використовувати функцію menu,
 яка повертає покажчик на функцію, що табулюється}
type
   TFile=file of Real;
   TFunc=function(a,b,c,x:Real):Real;
   TPtTFunc=^TFunc;
      {Визначення типу TPtTFunc для оголошення покажчиків...}
{...на функції з чотирма параметрами і результатом типу real}
{$F+}
function SinABC(a,b,c,x:Real):Real;
begin
   SinABC:= a*Sin(b*x+c);
end;
function ParabolaABC(a,b,c,x:Real):Real;
begin
   ParabolaABC:= a*(x+b)*(x+b)+c;
end;
{$F-}
procedure tabulation(d1,d2:Real;n:Integer;a,b,c:Real;
                     var fr:TFile;f:TFunc);
var
   dx,v:Real;
   i:Integer;
begin
   dx:=(d2-d1)/(n-1);
   WriteLn;
   for i:=1 to n do begin
     v:=f(a,b,c,d1+(i-1)*dx);
     Write(fr,v);
   end
end;
function menu:TPtTFunc;  {Визначення функції, що повертає...}
                                        {...покажчик на функцію}
const
   fun_items:array[1..2] of TPtTFunc =(@SinABC,@ParabolaABC);
                  {Визначений масив покажчиків на функції...}
                         {... і здійснена його ініціалізація}
var
   key:Integer;
begin
```

```
   WriteLn(#13#10'Для вибору функції введи 1 (синусоїда) ',
           #13#10'чи 2 (парабола)');
   ReadLn(key);
   case key of
    1,2:  menu:=fun_items[key];       {Повернення покажчика...}
                     {...на одну з функцій чи значення nil,...}
                              {... якщо невірно введений номер }
    else menu:=nil;
   end;
end;
var
   a,b,c,d1,d2:Real;
   n:Integer;
   FileOfReal:TFile;
   item:TPtTfunc;    {Змінна item визначається за допомогою...}
             {...раніше введеного типу TPtTfunc як покажчик...}
         {...на функцію з 4 параметрами і результатом типу real}
begin
   Assign(FileOfReal,'result.dat');
   Rewrite(FileOfReal);
   Write(#13#10'Введи коефіцієнти функції (a b c) ');
   ReadLn(a,b,c);
   Write('Введи межі табулювання d1 i d2 (d1<d2) ');
   ReadLn(d1,d2);
   Write('Введи кількість вузлів ');
   ReadLn(n);
   item:=menu;          {У item заноситься адреса функції чи nil}
   if item=nil then WriteLn('Номер функції введений невірно')
   else tabulation(d1,d2,n,a,b,c, FileOfReal,
                   TFunc(item));             {Використання item}
   Close(FileOfReal);
end.
```

Введений посилальний тип **TPtTFunc=^TFunc** з базовим функціональним типом **TFunc=function(a,b,c,x:Real):Real** використовується як тип значення, яке повертається функцією, що повертає покажчик на одну з функцій **SinABC** і **ParabolaABC**. Якщо функція, що табулюється, вибирається невірно, функція **menu** повертає порожній покажчик **nil**. Покажчик, повернутий функцією menu, дозволяє забезпечити виклик функції, на яку він вказує. Повернуте функцією menu значення записується в змінну **item**. Якщо в цій змінній не записане значення **nil**, то відповідний покажчик, зведений до типу **TFunc**, підставляється як фактичний параметр у підпрограму **tabulatio**n, забезпечуючи тим самим передачу у функцію **tabulation** адреси, за якою розташовується код функції, що табулюється.

6. МОДУЛІ

6.1. Призначення модулів і їхня структура

Коли говорять про файли, що містять програми на TP, мова йде про три типи файлів: 1) текстові файли з текстами програм на TP (звичайно ці файли мають розширення .pas, хоч це не обов'язково); 2) tpu-файли; 3) готові до виконання файли, отримані в результаті компіляції програм (exe-файли).

Що ж таке tpu-файли?

У міру розвитку обчислювальної техніки з'явилася можливість у масовому порядку вирішувати складні задачі, що вимагають написання великих програм із залученням для цього колективів розробників. Виникло поняття модульного програмування, під яким розуміють і розбивку програми на окремі фрагменти, і створення бібліотек фрагментів, з яких можна сформувати програму (у TP вставки в текст програми здійснюються за допомогою директиви компіляції **{$I** ім'я_файла_що_включається**}**), і написання підпрограм, у тому числі зовнішніх (ця можливість також є в TP). Можна також створювати програмні одиниці, що зберігаються і компілюються незалежно а від одної і мають певний інтерфейс, який дозволяє використовувати різні ресурси цих програмних одиниць у розроблювальних програмах. Такі програмні одиниці в TP називають модулями; в результаті їхньої компіляції створюються файли з розширенням .tpu (tpu-файли). Використання модулів дозволяє створювати великі програми (exe-файли), хоча розмір окремого tpu-файла не може перевищувати 64K.

У TP модуль (**unit**) вважається окремою програмою і створюється спочатку як звичайний pas-файл, що оформлюється за певними правилами:

```
unit ім'я_модуля;                    {Заголовок модуля}
interface          {Заголовок інтерфейсної частини}
uses список використовуваних модулів
type ...
const ...
var ...
procedure ...
function ...
```

implementation *{Заголовок розділу реалізації}*
Опис локальних модулів, типів, констант, змінних, процедур і функцій, а також процедур і функцій, заголовки яких оголошені в розділі **interface**
begin
Розділ ініціалізації
end.

Як і програма, текст модуля починається з заголовка і закінчується службовим словом **end** з крапкою. Однак заголовок модуля починається не зі слова **program**, а зі службового слова **unit** і є обов'язковим. Після слова **unit** вказується ім'я модуля, яке обов'язково повинно збігатися з ім'ям файла, що містить модуль (наприклад, для заголовка unit module текст модуля повинен зберігатися у файлі module.pas, у результаті компіляції якого створюється файл module.tpu). Завершується заголовок крапкою з комою. Наведене в заголовку ім'я модуля вказується при його підключенні до програми чи іншого модуля. Щоб підключити модуль до програми необхідно відразу ж слідом за заголовком програми (якщо він мається) після службового слова **uses** вказати ім'я модуля, що підключається (uses module). Службове слово **uses** може зустрітися в програмі тільки один раз, тому для підключення декількох модулів їхні імена перелічують через кому (uses module1,module2).

За заголовком модуля іде його інтерфейсна частина, яка починається зі службового слова **interface**. В ній перелічуються програмні ресурси (константи, типи, змінні, заголовки процедур і функцій), призначені для використання іншими модулями і програмами. Описані тут елементи називаються видимими поза модулем (зовнішніми). Вимоги до опису тут ті ж, що і взагалі в TP 7.0, але для процедур і функцій вказують тільки заголовки, причому описувачі **external**, **forward** і **assembler** не ставляться.

Відзначимо також, що в розділі **uses** перелічуються імена модулів, використовуваних у даному модулі. Однак усе-таки вони недосяжні програмам, що використовують даний модуль. Тому, щоб одержати в програмі доступ до ресурсів модуля, його потрібно оголосити явно, а не опосередковано (через інший модуль).

Слідом за інтерфейсною частиною модуля йде розділ реалізації, що починається зі службового слова **implementation**. Тут

оголошуються невідомі поза модулем його внутрішні елементи (їх ще називають невидимими, схованими), а саме, локальні змінні, константи, типи, процедури і функції, а також модулі, ресурси яких використовуються всередині даного модуля. Крім того, тут же здійснюється реалізація тих процедур і функцій, заголовки яких описані в інтерфейсній частині і які відомі поза модулем. Такий метод опису процедур і функцій обраний з такої причини: користувачу для звертання до підпрограми досить знати її ім'я і список параметрів. Знання техніки її реалізації зовсім не потрібне. Якщо здійснюється модифікація підпрограми, поміщеної в модуль, то досить змінити тільки її тіло в розділі реалізації, не змінюючи заголовка в інтерфейсній частині. В зв'язку з цим програми, що використовують цю підпрограму, не змінюються. Якщо підпрограма оголошена в інтерфейсній частині, то в розділі реалізації її заголовок звичайно дається тільки у вигляді імені без вказівки переліку і типів параметрів, а також типу значення, що повертається, для функції (аналогічно випереджальному опису).

Розділи **interface** і **implementation** обов'язкові; навіть якщо вони порожні, їхні заголовки вказати потрібно.

Після розділу реалізації в модулі розташовується розділ ініціалізації, який починається зі слова **begin**, за яким йдуть оператори, що будуть виконуватися до операторів з тіла програми (наприклад, установка початкових значень для деяких змінних, оголошених у модулі). Розділ ініціалізації не обов'язковий, тому, якщо при підключенні модуля не потрібно робити ніяких початкових установок, він може бути опущений (разом зі службовим словом **begin**).

6.2. Компіляція програм, що використовують модулі

До програми можна підключити кілька модулів. У свою чергу, модуль може використовувати ресурси інших модулів. У цьому випадку в програмі в специфікації **uses** вказують тільки імена тих модулів, які безпосередньо використовуються програмою. Характерною рисою модулів є те, що вони не цілком включаються в exe-файл: до програми додається з модуля тільки те, що використовується. Заборонено явне чи опосередковане звертання модуля до самого себе, хоча, якщо модуль з'являється в розділі реалізації, це обмеження поширюється тільки на пряме звертання.

Можливий випадок, коли в програмі й у інтерфейсних частинах декількох модулів, підключених до неї, оголошені за допомогою одного імені різні об'єкти (наприклад, у програмі є оголошення uses module1,module2 і як у програмі, так і в інтерфейсних частинах обох модулів є описи з ім'ям name). У цьому випадку програма оперує, насамперед, елементами, оголошеними в ній самій (тобто тим елементом name, що оголошений у програмі). Для звертання до об'єкта, ім'я якого перекрито, необхідно зліва до імені об'єкта дописати ім'я модуля, в якому він оголошений, з'єднавши їх крапкою (наприклад, module1.name чи **System.Real**).

У TP немає зв'язку між ім'ям програми й ім'ям файла, в якому вона зберігається. У той же час для підключення модуля до програми він повинен бути знайдений на диску. Тому ім'я модуля й ім'я файла, що містить модуль, повинні збігатися. Вихідний текст модуля зберігається у файлі з розширенням .pas, а отриманий у результаті компіляції код модуля – у файлі з розширенням .tpu (від Turbo Pascal Unit). Так, якщо ім'я модуля module, то відповідними файлами будуть module.pas і module.tpu.

При компіляції програми в режимі Compile/Compile компілятор послідовно відшукує tpu-файли, що містять коди використовуваних модулів, для їхнього підключення. Пошук проводиться в такий спосіб:

1. Перегляд системного бібліотечного файла модулів turbo.tpl (від Turbo Pascal Librery), який повинен зберігатися в тому ж каталозі, що і файл turbo.exe.

2. Якщо модуля немає у файлі turbo.tpl, то пошук продовжується в поточному каталозі.

3. Якщо і тут модуль не знайдений, то йде перегляд каталогів, перелічених в альтернативі Options/Directories/Unit Directories, з видачею діагностичного повідомлення і припиненням компіляції, якщо модуль не виявлений (перелік каталогів провадиться через крапку з комою).

При виборі альтернативи Compile/Make здійснюється пошук pas-файла, що містить вихідний текст модуля. У випадку його виявлення в тому ж каталозі шукається відповідний tpu-файл. Якщо tpu-файл створений пізніше pas-файла, то йде підключення модуля з tpu-файла. Якщо ж tpu-файл не виявлений чи pas-файл корегувався

після створення tpu-файла, то в обов'язковому порядку компілюється pas-файл з текстом модуля.

При компіляції в режимі Compile/Build в обов'язковому порядку йде компіляція всіх pas-файлів, що містять тексти використовуваних модулів. Якщо якого-небудь pas-файла нема, але є відповідний tpu-файл, здійснюється підключення останнього.

Модулі завжди компілюються перед компіляцією програми.

Системний бібліотечний файл модулів turbo.tpl (у TP він єдиний) має спеціальну структуру, орієнтовану на швидкий пошук модулів, що містяться в ньому. Включення в нього модулів і виключення їх виконуються за допомогою спеціальної програми tpumover.exe. Пошук модулів у файлі turbo.tpl здійснюється завжди, тому недоцільно робити його занадто великим, бо це збільшує час компіляції через необхідність перегляду великого файла. У той же час часто використовувані модулі, звичайно, потрібно поміщати у файл turbo.tpl.

6.3. Приклад оформлення модуля і його підключення

Розглянемо особливості реалізації і підключення модулів на конкретному прикладі.

```
{Приклад 6.1}
{Модуль для роботи з комплексними числами: визначення
 типу комплексного числа, констант, процедур завдання
 (InitComplexValue) і виводу (WriteLnComplexValue)
 комплексного числа, додавання (AddComplexValues),
 множення (MultComplexValues) і ділення (DivComplexValues)
 комплексних чисел}
unit Compl_Un;
interface                          {Початок інтерфейсної частини}
type
   Complex=record                                  {Тип комплексний}
             Re,Im:Real;
   end;
const
   ComplexZero:Complex=(Re:0;Im:0);          {Комплексний нуль}
   ImagineOne:Complex=(Re:0;Im:1);             {Уявна одиниця}
procedure InitComplexValue(r,i:Real;var c:Complex);
procedure WriteLnComplexValue(c:Complex);
procedure AddComplexValues(c1,c2:Complex;var result:Complex);
procedure MultComplexValues(c1,c2:Complex;var result:Complex);
procedure DivComplexValues(c1,c2:Complex;var result:Complex);
```

```
implementation                                    {Розділ реалізації}
procedure InitComplexValue;
begin
   with c do begin
      Re:=r;Im:=i
   end
end;
procedure WriteLnComplexValue;
begin
   with c do begin
      Write(Re);
      if Im>0 then Write('+');
      if Im<>0 then Write(Im,'i');
      WriteLn
   end
end;
procedure AddComplexValues;
begin
   with result do begin
      Re:=c1.Re+c2.Re;Im:=c1.Im+c2.Im
   end
end;
procedure MultComplexValues;
begin
   with result do begin
      Re:=c1.Re*c2.Re-c1.Im*c2.Im;
      Im:=c1.Re*c2.Im+c1.Im*c2.Re
   end
end;
procedure DivComplexValues;
var
   z:Real;
begin
   with c2 do z:=Re*Re+Im*Im;
   with result do begin
      Re:=(c1.Re*c2.Re+c1.Im*c2.Im)/z;
      Im:=(c2.Re*c1.Im-c1.Re*c2.Im)/z
   end
end;                              {Розділ ініціалізації відсутній}
end.
```

Розглянемо особливості наведеного тексту модуля:

1. Ім'я модуля складається з 8 символів. Більш довге ім'я використовувати не можна, бо воно повинно збігатися з ім'ям файла, а імена файлів не можуть містити більше 8 символів. Текст модуля повинен зберігатися у файлі Compl_Un.pas.

2. У інтерфейсній частині оголошений тип Complex і дві константи ComplexZero і ImagineOne; вони можуть бути використані як усередині модуля, так і в будь-якій програмі чи будь-якому модулі, до якого підключений модуль Compl_Un.

3. Оголошені в інтерфейсній частині п’ять процедур також можуть бути використані як усередині модуля, так і поза модулем.

4. У розділі реалізації вказані скорочені заголовки процедур, хоча їх можна повторити в тому ж вигляді, що й в інтерфейсній частині.

5. Розділ ініціалізації модуля відсутній, про що свідчить пропуск службового слова **begin**. У даному випадку крапка з комою перед **end** з крапкою обов’язкові (якби в модулі був присутній розділ ініціалізації, її довелося б поставити перед **begin**).

Для використання яких-небудь процедур роботи з комплексними числами досить у програмі в розділі **uses** оголосити модуль Compl_Un. Після цього з процедурами модуля можна працювати так, ніби вони були оголошені в самій програмі.

```
{Приклад 6.2}
{Вводяться два комплексних числа. Надрукувати їх
 добуток. Скористатися ресурсами модуля Compl_Un}
uses Compl_Un;                          {Підключення модуля}
var
   c,c1,c2:Complex;
   r,i:Real;
begin
   WriteLn('Введіть дійсну і уявну ');
   WriteLn('частини першого комплексного числа');
   ReadLn(r);ReadLn(i);
   InitComplexValue(r,i,c1);
   WriteLn('Те ж для другого комплексного числа');
   ReadLn(r,i);
   InitComplexValue(r,i,c2);
   MultComplexValues(c1,c2,c);
   Write('Добуток дорівнює ');
   WriteLnComplexValue(c)
end.
```

У даній програмі відомий тип Complex і процедури InitComplexValue, MultComplexValues і WriteLnComplexValue за рахунок оголошення модуля Compl_Un у **uses**-рядку. Тільки ці елементи модуля збільшують exe-файл, всі інші елементи (константи ComplexZero, ImagineOne, процедури AddComplexValues і DivComplexValues) у exe-файл не потрапляють.

6.4. Стандартні модулі

У TP є 8 стандартних модулів **(System, Dos, Crt, Printer, Overlay, Graph, Turbo3, Graph3**). Модуль **System** підключається автоматично і його не потрібно згадувати в розділі **uses**. Всі інші модулі необхідно оголошувати в програмі для забезпечення доступу до їхніх ресурсів.

У модулі **System** зберігаються типи, константи, змінні і підпрограми авторської версії мови, стандартної мови, а також ряд додаткових підпрограм. Відсутність цього модуля призводить до неможливості компіляції будь-якої програми.

Модуль **Crt** (див. главу 7) використовується при роботі з екраном у текстовому режимі, при читанні з клавіатури, а також для забезпечення деяких інших функцій.

Модуль **Dos** (див. главу 10) забезпечує звертання до засобів операційної системи MS DOS.

Модуль **Printer** дозволяє здійснювати вивід на принтер за допомогою процедур **Write** і **WriteLn**. Для цього оголошена файлова змінна **Lst** типу **Text**, що зв'язана з логічним пристроєм **prn**, у результаті чого після підключення модуля **Printer** вивід на принтер можна робити, наприклад, так:

```
WriteLn(Lst,'Здійснюється вивід на принтер');
```

Модуль **Graph** (див. главу 11) забезпечує роботу з екраном у графічному режимі.

Модуль **Overlay** служить для ефективного виконання великих програм. Модулі **Turbo3** і **Graph3** використовуються вкрай рідко і служать для забезпечення сумісності більш пізніх версій мови з програмами, написаними мовою TP версії 3.0.

Перші 5 згаданих модулів часто включають у файл turbo.tpl.

Існує також цілий ряд інших модулів, що стали практично стандартними. Насамперед, це модулі, які входять у бібліотеку Turbo Vision (Objects, Views, Dialogs, App і ін.) і призначені для побудови інтерфейсних частин розроблювальних програм.

Для роботи з довгими (довжиною більше 255 символів) рядками розроблений модуль **Strings** (див. главу 12).

7. МОДУЛЬ CRT

7.1. Загальні відомості про роботу з екраном. Текстові режими

Основним засобом відображення інформації, що вводиться і виводиться під час роботи прикладних і системних програм, є екран. TP дозволяє працювати з екраном у текстовому і графічному режимах.

Текстовий режим служить для відображення символів кодової таблиці і характеризується максимальним числом символів у рядку і кількістю рядків на екрані. Крім того, монохромні дисплеї характеризуються кількістю ступенів яскравості, а кольорові – можливою кількістю використовуваних кольорів. Мінімальною одиницею керування в текстових режимах є символ. Він складається з декількох пікселів (точок), перетворення яких відбувається на апаратному рівні. Для збереження виведеного на екран символу в текстових режимах потрібно 2 байти: перший містить безпосередньо символ кодової таблиці; другий вказує, як символ повинен бути виведений на екран (його колір, колір фону під символом, мерехтіння).

Засоби для роботи з екраном у текстовому режимі зосереджені в модулі **Crt**, який підключається звичайним чином:

```
uses Crt;
```

Установка текстового режиму здійснюється за допомогою процедури **TextMode**, виконання якої приводить до очищення екрана й активізації зазначеного режиму. У модулі **Crt** визначені константи для установки текстових режимів:

BW40 – чорно-біле зображення з екраном у 40 стовпців і 25 рядків (40×25) для кольорового монітора (BW40=0);

BW80 – те ж з екраном 80×25 (BW80=2);

CO40 – кольоровий текстовий режим 40×25 (CO40=1);

CO80 – те ж з екраном 80×25 (CO80=3);

C40=CO40 і **C80=CO80** – введені для сумісності з TP 3.0;

Mono – монохромний режим 80×25.

Для установки того чи іншого текстового режиму необхідно звернутися до процедури

TextMode(режим**)**

з вказівкою за параметр однієї з наведених вище констант. Наприклад, для установки кольорового режиму 40×25 можливий один з таких трьох варіантів звертання:

```
TextMode(CO40);     TextMode(C40);     TextMode(3);
```

Процедура **TextMode** може бути використана без параметра. У цьому випадку активізується режим, що зустрічався останнім до поточного режиму. За замовчуванням прийняті режими **BW80** і **C80**.

У модулі **Crt** є константа **Font8x8** зі значенням 256, що при додаванні до основної константи збільшує кількість рядків до 43 для адаптерів EGA і 50 для адаптерів VGA:

```
TextMode(CO80+Font8x8);
```

Зміна кількості рядків відбувається за рахунок установки розміру символів рівного 8×8 пікселів, замість стандартних шрифтів 8×14 для EGA і 8×16 для VGA.

Поточний текстовий режим зберігається у введеній у модулі **Crt** змінній **LastMode** типу **Word**. Вона використовується програмістом найчастіше при поверненні з графічного режиму в текстовий:

```
TextMode(LastMode);
```

7.2. Введення/вивід при підключеному модулі Crt

При підключенні модуля **Crt** значно підвищується швидкість виводу інформації на дисплей за рахунок підключення стандартних файлів **Input** і **Output** до фіктивного пристрою **Crt**. Тому, навіть якщо в програмі не використовуються процедури і функції з модуля **Crt**, його потрібно підключати. Це відбувається автоматично при виконанні оператора uses Crt. Якщо вводяться/виводяться файли, відмінні від стандартних, то для прискорення обміну їх можна зв'язати з пристроєм **Crt** процедурою **AssignCrt**, єдиний параметр якої – змінна типу **Text**.

```
{Приклад 7.1}
{Використання AssignCrt}
uses Crt;
var
  f:Text;
begin
  AssignCrt(f); {Файл f зв'язаний із пристроєм Crt і вико-...}
Rewrite(f);     {...ристовує його механізм для швидкого виводу}
  WriteLn(f,'Інформація');          {Швидкий вивід на монітор}
  Close(f)
end.
```

При використанні модуля **Crt** можна виводити на екран усі символи, у тому числі і керуючі (коди 0..31). Однак 4 символи не втрачають свій керуючий вплив:

#7 – короткий звук динаміка;

#8 – зрушення курсору на 1 позицію вліво (якщо це можливо);

#10 – зрушення курсору на рядок униз (якщо рядок останній, зображення зрушується на 1 рядок вгору);

#13 – переведення курсору в початок рядка.

Якщо модуль **Crt** не підключений, то при введенні даних з клавіатури можна редагувати рядок, що вводиться (до моменту натискання клавіші <Enter>), застосовуючи тільки клавішу <BackSpace>. Модуль **Crt** забезпечує можливість використання для редагування при введенні й інших клавіш та їх комбінацій:

<Enter> – завершує введення рядка;

<BackSpace> – видаляє останній введений символ;

<Esc> – видаляє весь рядок введення;

<Ctrl/S> – те ж, що і <Esc>;

<Ctrl/A> – те ж, що і <BackSpace>;

<Ctrl/D> – виводить на екран останній стертий символ;

<Ctrl/F> – відновлює на екрані весь раніше стертий рядок;

<Ctrl/Z> – завершує введення рядка і генерує маркер кінця файла (#26), якщо змінна **CheckEof** модулю **Crt** має значення **True**.

7.3. Вивід на кольоровий дисплей

Кожен виведений символ може мати один з 16 (від 0 до 15) можливих кольорів. Фоновий колір (колір позиції, в якій виводиться символ) вибирається з перших 8 значень (від 0 до 7). У модулі **Crt** для роботи з кольорами визначені такі 16 констант:

Black=0	– чорний,	**DarkGray**=8	– темно-сірий,
Blue=1	– синій,	**LightBlue**=9	– світло-синій,
Green=2	– зелений,	**LightGreen**=10	– світло-зелений,
Cyan=3	– голубий,	**LightCyan**=11	– світло-голубий,
Red=4	– червоний,	**LightRed**=12	– світло-червоний,
Magenta=5	– фіолетовий,	**LightMagenta**=13	– бузковий,
Brown=6	– коричневий,	**Yellow**=14	– жовтий,
LightGray=7	– світло-сірий,	**White**=15	– білий.

За замовчуванням колір для символів встановлюється з максимальним (**White**), для фону – з мінімальним (**Black**) номером.

Процедури

TextColor(номер_кольору**)**

і

TextBackground(номер_кольору**),**

що мають один параметр типу **Byte**, дозволяють здійснювати роздільне керування кольором символів і фону відповідно. Так оператори

```
TextColor(Yellow);TextBackground(Red);
```

встановлюють, що надалі вивід буде здійснюватися жовтим кольором на червоному фоні.

Можна забезпечити вивід за допомогою мерехтливих символів. Для цього визначена константа **Blink**=128, яка повинна бути додана до номера кольору в операторі **TextColor**:

```
TextColor(11+Blink).
```

Найчастіше доцільним є організація виводу на чистому екрані. Щоб очистити екран (залити його поточним кольором фону) використовують процедуру без параметрів **ClrScr**, що, крім того, переводить курсор у лівий верхній кут екрана (взагалі говорячи, ця процедура працює не з екраном, а з вікном).

Колірні атрибути виведених символів зберігаються у визначеній у модулі **Crt** типізованій константі **TextAttr**, що має тип **Byte** і стартове значення 15. Молодші 4 біти цієї змінної служать для збереження кольору символів (4 біти – 16 кольорів), наступні 3 біти – для збереження кольору фону (3 біти – 8 кольорів), старший біт – біт блінкування (якщо в ньому записана 1, то символи будуть мерехтіти). Треба пам’ятати, що для установки кольору фону в змінній **TextAttr** константа повинна бути помножена на 16. Так, наприклад, оператор

```
TextAttr:=LightRed+16*LightGray+Blink
```

встановлює подальший вивід світло-червоними символами на світло-сірому фоні з мерехтінням.

7.4. Позиціонування при виводі на екран і створення текстових вікон

Нумерація стовпців (позиція X) у текстовому режимі здійснюється зліва направо від позиції 1, а нумерація рядків (позиція Y) – зверху униз від рядка 1. Таким чином, лівий верхній кут екрана в текстовому

режимі має координати (1, 1). Оскільки вивід на екран проводиться, починаючи зі знакомісця, де розташовується курсор, керування виведенням пов'язано, насамперед, з керуванням положенням курсора, для чого в модулі **Crt** визначена процедура

GoToXY(номер_стовпця**,** номер_рядка**)**,

яка переміщує курсор у позицію з зазначеними координатами (тип **Integer**).

Функції без параметрів **WhereX** і **WhereY** повертають відповідно номер стовпця і номер рядка з поточним положенням курсора.

```
{Приклад 7.2}
{Вивести посередині ясно-сірого екрана жовтими
символами на червоному фоні заданий рядок}
uses Crt;
var
   s:string[80];   {Максимальна довжина рядка - ширина екрана}
begin
 TextAttr:=White+16*Black;          {Вивід білими символами...}
 ClrScr;                  {... на чорному фоні; очищення екрана}
 WriteLn('Введіть рядок');ReadLn(s);
 TextBackground(LightGray);
 ClrScr;                                    {Тепер екран ясно-сірий}
 TextBackground(Red);
 TextColor(Yellow);
 GoToXY((80-Length(s)) div 2+1,13);{Курсор у потрібній позиції}
 Write(s)
end.
```

Дотепер, говорячи про вивід на екран, ми мали на увазі розгляд повного екрана. Модуль **Crt** дозволяє створювати текстові вікна.

Вікно – це обмежена прямокутна область екрана зі сторонами, рівнобіжними його межам, яка виконує ті ж функції, що і повний екран. Для організації вікна використовується процедура **Window**, що має 4 параметри, які задають координати лівого верхнього (X1, Y1) і правого нижнього (X2, Y2) кутів вікна:

Window(X1**,**Y1**,**Y2**,**Y2**).**

Частина екрана поза вікном залишається недоступною до оголошення нового вікна, а процедури і функції для роботи з екраном починають працювати усередині вікна. Положення курсора при оголошенні вікна визначається системою координат, відлічуваною від лівого верхнього кута вікна. Слід зазначити, що координати вікна, яке оголошується, задаються в системі координат екрана, а не старого

вікна. Щоб повернутися до роботи на всьому екрані, досить оголосити вікно на весь екран:

```
Window(1,1,80,25).
```

Відзначимо також, що при оголошенні вікна ніяких змін, які можна побачити (очищення вікна, рамка і т.п.), не відбувається (за винятком того, що курсор переміщується в лівий верхній кут нового вікна).

При створенні поточного вікна повинні виконуватися умови 1<=X1<=X2<=max(X) і 1<=Y1<=Y2<=max(Y). Якщо яка-небудь з умов не виконується, то вікно не створюється. Залежно від режиму монітора max(X) приймає одне зі значень 40 і 80. Значення max(Y) на всіх моніторах, крім EGA і VGA, дорівнює 25. Для адаптерів EGA можна встановити значення max(Y)=43, а для адаптерів VGA і SVGA – 50.

Процедура **Window** заносить координати поточного вікна в змінні **WindMin** і **WindMax** типу **Word** (відповідно лівий верхній і правий нижній кути). Координата X запам'ятовується в молодшому байті, а координата Y – у старшому байті цих змінних. Запам'ятовані значення відраховуються не від 1, а від 0. Щоб довідатися про координати поточного вікна, необхідно скористатися функціями **Lo** і **Hi**:

Lo(WindMin)+1 – координата X лівого верхнього кута поточного вікна;

Hi(WindMax)+1 – координата Y правого нижнього кута поточного вікна.

За замовчуванням значення змінних **WindMin** і **WindMax** відповідають розміру екрана (див. текстові режими). Якщо за допомогою процедури **TextMode** встановлений новий текстовий режим, то відповідним чином змінюються і значення змінних **WindMin** і **WindMax**. Так, наприклад, при виконанні оператора

```
TextMode(CO40+Font8x8);
```

у ці змінні будуть занесені значення такі, що Lo(WindMin)=0, Hi(WindMin)=0, Lo(WindMax)=39, Hi(WindMax)=50 (для VGA-адаптера).

```
{Приклад 7.3}
{Розфарбувати екран вертикальними смугами шириною в 2 позиції}
uses Crt;
var
  j:Byte;
```

```
begin
  TextBackground(Blue);
  ClrScr;                                          {Очищення екрана}
  TextAttr:=16*Red;                                 {Те ж, що і... }
                        {...TextBackground(Red);TextColor(Blak);}
  for j:=1 to 40 do     {Всього 40 колонок шириною в 2 позиції}
    if Odd(j) then begin        {Перефарбувати тільки смуги...}
       Window(2*j-1,1,2*j,25);          {...з парними номерами}
       ClrScr;                                      {Очищення вікна}
    end;
end.
```

7.5. Деякі процедури модуля Crt для роботи з екраном

Наступні три процедури без параметрів також можуть бути використані при написанні програм (діють у поточному вікні):

ClrEol – стирає символи від позиції курсора до кінця рядка без переміщення курсора;

DelLine – видаляє рядок, в якому знаходиться курсор, з переміщенням на позицію вгору усіх рядків нижче вилученого й очищенням нижнього рядка вікна;

InsLine – вставляє порожній рядок у позицію курсора з переміщенням поточного рядка і всіх рядків нижче його на позицію вниз.

Часто при виводі інформації на екран потрібно затримати на якийсь час виконання програми. У модулі **Crt** оголошена процедура

Delay(затримка**),**

аргументом якої є ціле значення (тип **Byte**), що задає час затримки в мілісекундах.

```
{Приклад 7.4}
{Зобразити в нижній частині екрана червоно-синій прямокутник з
 переходом від одного кольору до іншого по діагоналі від ліво-
 го верхнього кута до правого нижнього кута. Забезпечити пере-
 міщення його нагору до досягнення верхньої межі екрана}
uses Crt;
var
  i:Byte;
begin
  TextAttr:=White+16*Black;{Колірні атрибути за замовчуванням}
  ClrScr;
  Window(33,20,38,25);
  TextBackground(Red);
```

```
  ClrScr;  {Очищення вікна. Тут – заливання червоним кольором}
  TextBackground(Blue);                {Установка синього фону}
  for i:=1 to 5 do begin                {Переміщуємо курсор...}
    GotoXY(i+1,i);                  {...по діагоналі вікна і...}
    ClrEol;  {...заливаємо частину рядка вікна синім кольором}
  end;
  TextBackground(Black);             {Відновлення чорного фону}
  Window(1,1,80,25); {Вікно на весь екран. Курсор в лівому...}
  for i:=19 downto 1 do begin               {...верхньому куті}
     Delay(500);                                {Затримка 0.5 с}
     DelLine  {Видалення першого рядка і підйом зображення}
  end
end.
```

У модулі **Crt** оголошені ще три процедури без параметрів для роботи з екраном:

LowVideo – установити низьку інтенсивність символів, що виводяться;

HighVideo – установити високу інтенсивність символів, що виводяться;

NormVideo – установити нормальні колір і інтенсивність символів, що виводяться.

7.6. Системні змінні модуля Crt

При підключенні модуля **Crt** одержують свої значення визначені в ньому системні константи і змінні. Константи використовуються при звертанні до процедур модуля, а системні змінні обробляються засобами **Crt** і застосовуються як перемикачі режимів роботи механізмів введення/виводу модуля.

Змінні **TextAttr**, **LastMode**, **WindMin**, **WindMax** були розглянуті раніше. Зі змінними **DirectVideo** і **CheckSnow** при програмуванні зіштовхуються досить рідко, і ми їх розглядати не будемо.

Дві змінні, що зосталися (**CheckBreak** і **CheckEof**), при розв'язанні деяких задач використовуються програмістами.

Якщо модуль **Crt** не підключений, то натискання Ctrl/Z (символ з кодом 26) при введенні означає генерацію ознаки кінця файла і припиняє введення. У модулі **Crt** є можливість вмикати і вимикати інтерпретацію символу #26. Змінна **CheckEof** типу **Boolean** має стартове значення **False**. Якщо CheckEof=False, то натискання Ctrl/Z при введенні даних з клавіатури забезпечує генерування символу #26

без керуючого впливу. Якщо ж CheckEof=True, то символ #26 інтерпретується як кінець файла.

```
{Приклад 7.5}
{Введення даних із клавіатури до їхнього вичерпання з записом
 у текстовий файл f.txt}
uses Crt;
var
  s:string[50];                 {У рядку не більше 50 символів}
  f:Text;
begin
  ClrScr;
  Assign(f,'f.txt');Rewrite(f);
  WriteLn('Вводіть текст. Ознака закінчення введення Ctrl/Z');
  CheckEof:=True;                           {Інтерпретувати #26}
  while not Eof do begin   {Читати до кінця файла (до Ctrl/Z)}
     ReadLn(s);
     WriteLn(f,s);
  end;
  CheckEof:=False;             {Скасувати інтерпретацію Ctrl/Z}
  Close(f)
end.
```

У програмі використана функція **Eof** без параметра, що означає перевірку введення ознаки кінця файла з поточного пристрою (клавіатури).

Булевська змінна **CheckBreak** має стартове значення **True**, при якому натискання Ctrl/Break під час введення/виводу перериває роботу програми. При CheckBreak=False відбувається відключення механізму переривання програми за Ctrl/Break.

7.7. Звукові можливості модуля Crt

У модулі **Crt** визначені дві процедури (**Sound** і **NoSound**) для роботи з внутрішнім гучномовцем.

Процедура

Sound(частота**)**

з єдиним параметром типу **Word** включає гучномовець для генерації безупинного звукового сигналу з частотою (у Герцах), заданою значенням параметра. При цьому звук не відключається навіть при припиненні виконання програми.

Процедура без параметрів **NoSound** виключає внутрішній динамік.

```
{Приклад 7.6}
{Програти малу октаву. Тривалість звучання нот - 1 с}
uses Crt;
const      {Задаємо частоти нот до, ре,...,сі в малій октаві}
  Hz_Note:array[1..7] of Real=
          (130.81,146.83,164.81,174.61,196.00,220.00,246.94);
var
  i:Byte;
begin
  for i:=1 to 7 do begin
    Sound(Round(Hz_Note[i]));
    Delay(500);
  end;
  NoSound
end.
```

7.8. Робота з клавіатурою

Основним засобом введення інформації є клавіатура. З погляду кодів, що повертаються, вона складається тільки з алфавітно-цифрових і функціональних клавіш (є ще керуючі клавіші Ctrl, Alt, Shift, які не мають самостійної дії). При натисканні будь-якої клавіші в мікропроцесор клавіатури посилається так званий код сканування, що залежить від місця розташування клавіші на клавіатурі і є унікальним для кожної клавіші. Алфавітно-цифрові клавіші повертають код з одного байта, що являє собою одне значення, функціональні клавіші повертають двохбайтовий код, що складається з двох значень, перше з яких завжди дорівнює нулю, а друге служить для упізнання клавіші.

Спочатку код клавіші потрапляє в буфер клавіатури, обсяг якого дорівнює 15 байт (символів), і тільки з нього можна зчитувати інформацію для подальшої обробки.

При аналізі натискання клавіші можна використовувати алгоритм:

1. Якщо буфер клавіатури порожній, то закінчити аналіз, бо жодна клавіша не натискалася.

2. Якщо в буфері клавіатури існує вміст, то зчитати значення коду з буфера клавіатури.

3. Якщо зчитане значення відмінне від нуля, упізнати алфавітно-цифрову клавішу і закінчити аналіз.

4. Якщо зчитаний нуль, то клавіша функціональна.

5. Зчитати другу частину коду і упізнати функціональну клавішу.

Таким чином, при аналізі натискання функціональної клавіші повинне бути організоване дворазове зчитування коду.

При аналізі типу клавіш можна користатися визначеною в модулі **Crt** функцією без параметрів

ReadKey,

яка забезпечує зчитування символу з буфера клавіатури без його відображення на екрані. Якщо буфер клавіатури порожній, то йде очікування натискання клавіші; у противному разі перший символ, що міститься в буфері, передається в програму через ім'я функції, після чого вміст буфера зсувається на одну позицію.

```
{Приклад 7.7}
{Програма, яка пізнає тільки стрілки керування курсором
 і завершує роботу при натисканні клавіші <Esc>}
uses Crt;
var
  c:Char;
begin
  c:=ReadKey;
  repeat
    if c=#0 then  begin                {Перевірка старшого байта}
     c:=ReadKey;  {Зчитуємо другу половину коду функц. клавіші}
     case c of                        {Перевірка молодшого байта}
        #72:WriteLn('Вгору');
        #75:WriteLn('Уліво');
        #77:WriteLn('Вправо');
        #80:WriteLn('Униз')
     end;
    end;
    c:=ReadKey;
  until c=#27;
end.
```

Ще однією функцією, використовуваною при роботі з клавіатурою, є функція **KeyPressed**, яка не має параметрів. Вона повертає булівське значення **True**, якщо на клавіатурі була натиснута клавіша, і **False** у противному разі. Слід зазначити, що функція **KeyPressed** не змінює вміст буфера клавіатури, а здійснює тільки перевірку його непорожнечі.

Наступний приклад ілюструє роботу цієї функції:

```
WriteLn('Для продовження натисніть будь-яку клавішу');
repeat until KeyPressed;
```

Якщо програма запускається із середовища TP, то по закінченні роботи вміст буфера клавіатури друкується в редакторському вікні.

Тому наприкінці програми рекомендується здійснювати очищення буфера клавіатури, що можна зробити, наприклад, так (ch – символьна змінна):

```
while KeyPressed do ch:=ReadKey;
```

Відзначимо також, що при введенні даних з клавіатури оператори **Read** і **ReadLn** очищують буфер.

```
{Приклад 7.8}
{Без відображення на екрані ввести символ, що відповідає натис-
 нутій символьній клавіші. За допомогою стрілок перемістити
 курсор і після натискання <Enter> вивести символ на екран у
 позицію курсору. Процес повторювати до натискання <Esc>}
uses Crt;
var
  c1,c2:Char;
begin
  ClrScr;
  c1:=ReadKey;
  while c1<>#27 do begin              {Цикл до натискання <Esc>}
    if c1=#0 then c1:=ReadKey{Зчит. символу для функц. клавіші}
      else                           {Клавіша алфавітно-цифрова}
      repeat
        c2:=ReadKey;
        case c2 of
          #13:Write(c1);           {Друк при натисканні <Enter>}
          #0 :begin          {Якщо клавіша функціональна, то...}
               c2:=ReadKey; {...зчитати другу половину коду...}
               case c2 of    {...і переміщати курсор стрілками}
                #72:if WhereY<>1 then GotoXY(WhereX,WhereY-1);
                #75:if WhereX<>1 then GotoXY(WhereX-1,WhereY);
                #77:if WhereX<>80 then GotoXY(WhereX+1,WhereY);
                #80:if WhereY<>25 then GotoXY(WhereX,WhereY+1);
               end;
             end;
        end
      until c2=#13;          {Поки не натиснута клавіша <Enter>}
    c1:=ReadKey;                        {Зчитати нову клавішу}
  end;
  while not KeyPressed do;{Затримка до натиск. будь-якої клав.}
  while KeyPressed do c1:=ReadKey; {Очищення буфера клавіатури}
end.
```

Крім демонстрації методики використання функцій **ReadKey** і **KeyPressed**, у програмі показано, як можна звертатися до функцій **WhereX** і **WhereY**.

8. ЕЛЕМЕНТИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

8.1. Визначення об'єктів

Основою об'єктно-орієнтованого програмування (ООП) є ідея об'єднання в одній структурі даних і дій, які провадяться над цими даними (у термінах ООП — це методи).

Об'єкт — це структура, яка поєднує в собі дані різних типів (поля) і процедури та функції (методи), що їх використовують.

ООП базується на трьох поняттях: інкапсуляції, спадкуванні і поліморфізмі.

Інкапсуляція — це комбінування даних з методами, що обробляють ці дані, результатом чого є новий тип даних – об'єкт (**object**).

Спадкування — це можливість використання раніше визначених об'єктів для формування нових об'єктів («спадкоємців»), які успадковують описи даних «прабатька» та доступ до його методів.

Поліморфізм — це можливість визначення єдиної по імені дії (процедури чи функції), застосовної одночасно до всіх рівнів ієрархії спадкування. При цьому кожний об'єкт ієрархії може вказувати особливості дії над самим собою.

Структура опису даного типу є аналогічною структурі запису:

```
type
  ім'я_об'єкта=object
        поля;
        заголовки_методів;
end;
```

В описі типу об'єкта вказують тільки заголовки методів, а їхня реалізація здійснюється пізніше аналогічно тому, як описуються звичайні процедури і функції (при цьому перед ім'ям методу необхідно вказати ім'я типу об'єкта і крапку). Перелік параметрів при реалізації методів вказувати не обов'язково.

При оголошенні типізованої константи для типу **object** вказують тільки значення полів (за аналогією з записами).

Змінні об'єктового типу називають екземплярами об'єкта. Виклик методів об'єкта здійснюється з вказівкою імені екземпляра об'єкта.

```
{Приклад 8.1}
{Об'єкт виводить текстовий рядок у заданих координатах}
uses Crt;
type
   t1=object                          {Опис об'єктового типу}
         s:string;                                      {Поле}
         procedure Init(st:string);                   {Метод}
         procedure Run;                               {Метод}
   end;
var
   ob1:t1;                          {Опис екземпляра об'єкта}
   ss:string;
procedure t1.Init;                {Визначення методу t1.Init}
begin
  s:=st                             {Методу "відомо" поле s}
end;
procedure t1.Run;                  {Визначення методу t1.Run}
begin
 GotoXY(40,1);
 Write(s);
end;
begin
  ClrScr;
  WriteLn ('Введіть рядок');
  ReadLn(ss);
  ClrScr;
  ob1.Init(ss);                   {Звертання до методу Init}
  ob1.Run;                         {Звертання до методу Run}
end.
```

Як і записи, екземпляри об'єктів можуть брати участь в операторі **with**:

```
with ob1 do begin
  Init(ss);
  Run
end;
```

Обмеження:

- поля даних повинні стояти перед заголовками методів;
- поля об'єктів не можуть мати файлового типу;
- файли не можуть містити компоненти типу об'єкт;
- описи типу **object** не можуть зустрічатися локально усередині підпрограм;
- імена формальних параметрів методів не можуть збігатися з іменами полів.

Методи об’єкта оперують, насамперед, його полями, але можуть маніпулювати й іншими змінними, описаними в програмі. Однак об’єктний підхід припускає, що вся інформація про деякий фізичний об’єкт міститься у відповідному об’єкті, оголошеному в програмі. Тому при використанні об’єктів бажано писати програму так, щоб методи, що оголошені в об’єкті, зверталися тільки один до одного і до полів даних цього ж об’єкта. Аналогічно прагнуть до того, щоб звертання до полів об’єкта здійснювалося тільки за допомогою методів цього об’єкта (хоч конструкції типу ReadLn(ob1.ss) припустимі).

Доступ методів до полів об’єкта здійснюється за рахунок того, що при виклику методу в нього, крім фактичних параметрів, передається невидимий параметр **Self** («свій»), що вказує, якому об’єкту належить метод. Параметр **Self** неявно приєднується до всіх імен, які збігаються з іменами полів і методів відповідного об’єкта. Наприклад, у методі Init у наведеній вище програмі оператор s:=st сприймається так, ніби він був записаний у вигляді Self.s:=st. Параметр **Self** у звичайних умовах не вказують, але його можна записувати й у явному вигляді (хоч це буває досить рідко).

8.2. Спадкування і перевизначення. Об’єкти і модулі

Будь-який об’єкт може породити похідний об’єктовий тип, який успадковує всі поля і методи прабатьківського типу і може мати свої власні нові поля і методи. Нові методи просто додаються. Якщо ім’я методу, оголошеного в об’єкті-нащадку, збігається з ім’ям якогось методу прабатьківського типу, то відбувається перевизначення цього методу. У цьому випадку об’єкту-нащадку відомий метод, який оголошений усередині нього самого. При перевизначенні методів до всіх нащадків переходить перевизначений метод, і він дійсний, поки не буде ще раз перевизначений на якому-небудь рівні ієрархії успадкування.

Для визначення типу об’єкта, як похідного від вже існуючого, необхідно після слова **object** у дужках вказати ім’я прабатьківського типу:

```
type
  ім’я_об’єкта-спадкоємця=object(ім’я_об’єкта-предка)
                нові_поля_об’єкта-спадкоємця;
```

нові_методи_об’єкта-спадкоємця**;**

end;

Рекомендується завжди починати з визначення елементарного рівня даних і методів, переходячи до більш складних понять за допомогою переходу від рівня до рівня в ієрархії спадкування.

Правилом гарного тону в ООП є таке: якщо в похідному типі описана нова процедура ініціалізації (тобто виконання початкових дій з екземпляром об’єкта, наприклад надання початкових значень полям), то в ній на самому початку повинна викликатися процедура ініціалізації безпосереднього прабатька. Це гарантує заповнення всіх полів, які повинні одержати свої значення при ініціалізації.

```
{Приклад 8.2}
{Об’єкт-предок виводить рядок у заданих координатах поточним
 кольором. Об’єкт-спадкоємець забезпечує вивід рядка заданим
 зовні кольором у першому рядку екрана з автоматичним
 переміщенням введеного рядка вниз до зникнення з екрана}
uses Crt;
type
   t1=object                              {Опис об’єктового типу}
         s:string;
         procedure Init(st:string);
         procedure Run;
   end;
procedure t1.Init;
begin
  s:=st
end;
procedure t1.Run;
begin
  ClrScr;
  GotoXY(40,13);
  Write(s);
end;
type
   t2=object(t1)                    {Опис "нащадка" для типу t1}
         col:Byte;                                    {Нове поле}
         procedure Init(st:string;c:Byte);     {Перевизначення}
         procedure Run;                        {Перевизначення}
   end;
procedure t2.Init;
begin
  t1.Init(st);              {Виклик процедури ініціалізації...}
  col:=c                  {...об’єкта-предка; те ж, що і s:=st}
end;
```

```
procedure t2.Run;
var
 i:Byte;
begin
  TextColor(col);
  ClrScr;
  GotoXY(40,1);Write(s);
  for i:=1 to 25 do
  begin
   Delay(500);InsLine
  end;
end;
var
   ob1:t1;                                    {Статична змінна}
   ob2:^t2;                                   {Динамічна змінна}
   ss:string;
   c:Byte;
begin
  ClrScr;
  WriteLn('Введіть рядок');ReadLn(ss);
  WriteLn('Переміщення є? - Y/N');
  case ReadKey of
  'y','Y': begin
            WriteLn('Введіть номер кольору');ReadLn (c);
            New(ob2);
            ob2^.Init(ss,c);ob2^.Run;
            Dispose(ob2);
           end;
  else     begin
            ob1.Init(ss);ob1.Run
           end
  end;
  repeat until KeyPressed;
end.
```

У TP 7.0 для виклику методу предка введено слово **inherited**, що вказується перед ім'ям методу замість імені предка і крапки. Наприклад, визначений вище метод t2.Init може бути описаний так:

```
procedure t2.Init;
begin
  inherited Init(st);                {Те ж, що і t1.Init(st);}
  col:=c
end;
```

Екземпляри об'єктів можуть бути як статичними змінними, так і динамічними. Як це показано в наведеній вище програмі, при використанні динамічних об'єктів можна застосувати звичайну техніку роботи з динамічними змінними. Однак процедури **New** і

Dispose, крім звичайного, мають розширений синтаксис для динамічних об’єктів (див. п.8.3).

Особливістю змінних типу **object** є те, що їм можна присвоювати як значення відповідного типу, так і значення будь-якого похідного типу, тобто спадкоємець може передати записані в ньому значення предку (але не навпаки).

Нехай для визначених у попередній програмі типів **t1** та **t2** оголошені змінні:

```
var
  s1_stat:t1;
  s2_stat:t2;
  s1_din:^t1;
  s2_din:^t2;
```

Тоді правильними є такі оператори:

```
s1_stat:=s2_stat;
s1_din^:=s2_din^;
s1_din:=s2_din;
```

Перші два оператори заповнюють усі поля екземпляра об’єкта-предка (у конкретному випадку одне поле s) значеннями, занесеними у відповідні поля екземпляра об’єкта-спадкоємця; поля об’єкта-спадкоємця, які не існують в об’єкта-предка, не обробляються. Третій оператор забезпечує передачу посилання на екземпляр об’єкта-спадкоємця. При такому способі присвоювання всі поля екземпляра об’єкта, що стоїть у лівій частині оператора присвоювання, гарантовано будуть заповнені.

Методи при присвоюванні об’єктів не передаються.

Якщо в наведених вище операторах присвоювання поміняти місцями ліву і праву частини, то компілятор видасть помилку.

Змінні типу **object** можуть використовуватися як параметри процедур і функцій. При цьому діє те ж правило: фактичний параметр може мати не тільки той же тип, що і формальний параметр, але і будь-який похідний від нього тип. За рахунок такої властивості сумісності об’єктів забезпечується реалізація поліморфізму, результатом чого є така перевага: не потрібно піклуватися про те, щоб підпрограма викликала різні однойменні методи, властиві об’єктам різних рівнів спадкування; це забезпечується автоматично за рахунок передачі відповідного фактичного параметра (правда, методи, що при цьому викликаються, повинні бути віртуальними чи динамічними).

Властивість спадкування, притаманна об'єктам, визначає їхнє використання в модулях. При цьому в інтерфейсній частині модуля дається опис типу об'єкта, а реалізація методів здійснюється в розділі реалізації. Для використання описаних таким способом об'єктів досить знати тільки інтерфейсну частину модуля. Якщо властивості оголошеного в модулі об'єкта не задовольняють програміста, необхідно у своїй програмі оголосити об'єкт-спадкоємець, додавши нові методи і перевизначивши ті з методів прабатьківського типу, що не потрібні нащадку.

8.3. Віртуальні методи. Конструктори і деструктори

Розв'яжемо таку задачу. Описати в модулі два об'єкти: на верхньому рівні – нерухомий зафарбований прямокутник; спадкоємець – зафарбований прямокутник, що переміщується вниз при натисканні стрілки «униз» з відображенням координат його лівого верхнього кута в першому рядку екрана. В основній програмі визначити аналогічний об'єкт-прямокутник, але координати його лівого верхнього кута повинні відображатися усередині.

Текст модуля для даної задачі може бути, наприклад, таким:

```
unit BarUnit;
interface
uses Crt;
type
   TBarCrt=object                          {Нерухомий прямокутник}
               x1,y1,x2,y2:Integer;
               Color:Byte;
               constructor Init(xp1,yp1,xp2,yp2:Integer;c:Byte);
               destructor Done;
               procedure SwitchBar;
               procedure Run;virtual;
   end;
   TMoveBarCrt=object(TBarCrt)          {Може переміщатися вниз}
                  procedure WriteCoord;virtual;
                  procedure Move;
                  procedure Run;virtual;
   end;
implementation
constructor TBarCrt.Init;
begin
   x1:=xp1;y1:=yp1;x2:=xp2;y2:=yp2;Color:=c
end;
procedure TBarCrt.SwitchBar;         {Висвічування прямокутника}
```

```
begin
   Window(x1,y1,x2,y2);
   TextBackground(Color);ClrScr
end;
procedure TBarCrt.Run;          {Дії з нерухомим прямокутником}
var ch:Char;
begin
   SwitchBar;
   while ch<>#27 do ch:=ReadKey;
end;
procedure TMoveBarCrt.WriteCoord;          {Друк координат...}
begin                             {...переміщуваного прямокутника}
   TextColor(Yellow);
   GotoXY(1,1);Write('x:',x1:2,'; y:',y1:2)
end;
procedure TMoveBarCrt.Move;          {Переміщення прямокутника}
begin
   if y2<25 then begin             {Якщо не досягнений 25-й...}
     Window(1,1,80,25);            {... рядок, переміщати вниз}
     TextBackground(Black);
     GotoXY(x1,y1);InsLine;
     y1:=y1+1;y2:=y2+1;
     WriteCoord;
   end else Write(#7)          {Інакше видати звуковий сигнал}
end;
procedure TMoveBarCrt.Run;             {Дії з переміщуваним...}
var                                          {...прямокутником}
  ch:Char;
begin
   SwitchBar;
   ch:=ReadKey;
   while ch<>#27 do begin
     if ch=#0 then begin
       ch:=ReadKey;
       if ch=#80 then Move
     end;
     ch:=ReadKey
   end;
end;
destructor TBarCrt.Done;
begin                                    {Тільки стандартні дії}
end;
end.
```

Текст основної програми такий:

```
uses Crt,BarUnit;
type
   TTextBarCrt=object(TMoveBarCrt)     {Прямокутник з текстом}
                procedure WriteCoord;virtual;
```

```
   end;
procedure TTextBarCrt.WriteCoord;          {Друк координат...}
begin                                  {...прямокутника з текстом}
   TextBackground(Black);
   Window(x1,y1,x2,y2);
   TextColor(Yellow);
   GotoXY(4,(y2-y1)div 2+1);
   Write('x:',x1:2,'; y:',y1:2)
end;
var
   PBarCrt:^TBarCrt;
   PMoveBarCrt:^TMoveBarCrt;
   PTextBarCrt:^TTextBarCrt;
   ch:Char;
begin
   TextBackground(Black);ClrScr;
   Write('Move? Y/N');ch:=ReadKey;
   ClrScr;
   if ch in ['Y','y'] then begin
      Write('Bar & Text? Y/N');ch:=ReadKey;
      ClrScr;
      if ch in ['Y','y'] then begin
         New(PTextBarCrt,Init(20,1,40,7,Blue)); {Розширений...}
         PBarCrt:=PTextBarCrt;                 {...синтаксис New}
      end
      else begin
         New(PMoveBarCrt);{Звичайний синтаксис New з наступною}
         PMoveBarCrt^.Init(20,1,40,7,Blue); {...ініціалізацією}
         PBarCrt:=PMoveBarCrt;
      end
   end
   else begin
      New(PBarCrt,Init(20,1,40,7,Blue));
   end;
   PBarCrt^.Run;              {Викликається той метод Run, що...}
      {...відповідає екземпляру об'єкта, записаному в PBarCrt^}
   Dispose(PBarCrt,Done){Аналогічно здійсн. руйнування об'єкта}
end.
```

Насамперед, відзначимо, що в модулі й у програмі зустрілися нові службові слова (**virtual, constructor, destructor**), а також використані нові формати операторів **New** і **Dispose**:

New(ім'я_екземпляра_об'єкта**,** ім'я_конструктора**)**

і

Dispose(ім'я_екземпляра_об'єкта**,** ім'я_деструктора**).**

Методи, слідом за заголовком яких використане службове слово **virtual**, називаються віртуальними, на відміну від інших методів, що є статичними. У чому їхня відмінність і що забезпечує віртуалізація методів?

Якщо метод статичний, то при компіляції з ним жорстко зв'язуються всі методи, які звертаються до нього та належать до того об'єкта, який містить цей метод (оскільки вони усі компілюються в одному контексті). При цьому, якщо спадкоємець викликає успадкований від предка метод, то цей метод завжди буде викликати методи, визначені для предка, навіть якщо в спадкоємця є однойменні методи, які фактично необхідно викликати.

Проілюструємо сказане на прикладі наведеної програми. Для цього опустимо службове слово **virtual** в описі методів WriteCoord в об'єктів TMoveBarCrt і TTextBarCrt, зробивши їх статичними. При виклику методу Move усередині процедури TMoveBarCrt.Run спочатку відбудеться звертання до коду методу TMoveBarCrt.WriteCoord, у результаті чого текст буде виводитися у верхньому рядку екрана. Виконання ж методу Move для екземпляра об'єкта типу TTextBarCrt забезпечить виклик успадкованого методу TMoveBarCrt.Move, який, будучи жорстко зв'язаним з методом TMoveBarCrt.WriteCoord, викликає саме цей метод, а не метод TTextBarCrt.WriteCoord, як це повинно бути для правильної роботи програми. В результаті координати будуть друкуватися знову в першому рядку екрана.

Щоб можна було використовувати один метод Move для різних об'єктових типів, потрібно розірвати його статичний зв'язок з методом TMoveBarCrt.WriteCoord і забезпечити для Move можливість виклику методу TMoveBarCrt.WriteCoord чи TTextBarCrt.WriteCoord залежно від того, який об'єкт викликає метод Move. Такий механізм називається динамічним чи пізнім зв'язуванням на відміну від статичного чи раннього зв'язування. Він реалізується за рахунок введення віртуальних методів.

Використання віртуальних методів передбачає виконання таких умов:

1. Якщо в об'єкта-предка який-небудь метод описаний як віртуальний, то у всіх об'єктів-спадкоємців однойменні методи повинні бути також віртуальними. При цьому повинні залишатися

незмінними порядок розташування, кількість і типи параметрів в однойменних віртуальних методів.

2. В описі об'єкта, що має віртуальні методи, обов'язково повинен бути спеціальний метод, в оголошенні і реалізації якого службове слово **procedure** замінене словом **constructor**. Цей метод є особливим видом процедури (конструктором), призначеної для виконання настановних дій для забезпечення механізму віртуальних методів. Конструктор може успадковуватися, але не може бути віртуальним. Конструктор завжди викликається до першого виклику віртуального методу для даного екземпляра об'єкта. Один об'єкт може містити декілька конструкторів.

Для динамічних об'єктів введена розширена процедура **New**, першим параметром якої є ім'я посилання на об'єкт, а другим – ім'я відповідного конструктора (у програмі наведені два варіанти створення й ініціалізації динамічних об'єктів). Як і для звичайних динамічних змінних, для динамічних об'єктів може бути використана функція **New**, в якої також є розширений вид. У цьому випадку першим параметром є тип посилання на об'єкт (а не екземпляр об'єкта, як це має місце у випадку застосування процедури **New**).

```
{Приклад 8.3}
type
   TOb=object
         a:Real;
         constructor Init(b:Real);
           {. . .}
   end;
    NamePtr=^TOb;
constructor TOb.Init;
begin
end;
var
   ObPtr:NamePtr;
begin
   ObPtr:=New(NamePtr,Init(1.2));        {Функція New}
     {Еквівалентом є такі два оператори...}
     {ObPtr:=New(NamePtr);ObPtr^.Init(1.2);}
           {Продовження програми}
end.
```

Для виконання завершальних дій з об'єктами введений спеціальний метод – деструктор, що оформляється так само, як метод-процедура із заміною службового слова **procedure** службовим словом **destructor**. Цей метод служить для правильного звільнення

пам’яті, що займають динамічні об’єкти. Процедура **Dispose** знищує об’єкт у цілому. Але якщо поля об’єкта були динамічними і створювалися при виконанні конструктора, то їх треба звільнити до знищення об’єкта. Ці дії виконуються в деструкторах. Якщо об’єкт містить віртуальні методи, то деструктор правильно визначає його розмір, який повинний бути переданий процедурі **Dispose**. У зв’язку з цим для об’єктів введений розширений синтаксис процедури **Dispose** (перший параметр – ім’я посилання на об’єкт, другий – ім’я деструктора). При цьому спочатку виконується деструктор, а потім власне процедура **Dispose**.

Деструкторів може бути декілька в одного об’єкта, вони можуть успадковуватися і бути віртуальними. Більш того, рекомендується описувати деструктори віртуальними, щоб гарантувати виклик деструктора саме того об’єкта, що знищується.

8.4. Сховані і відкриті поля і методи

Частина полів і методів об’єктових типів можна оголосити схованими. Сховані компоненти об’єктів вважаються відомими тільки в межах програми чи модуля, у якому даний опис об’єктового типу. В інших модулях вони, на відміну від інших компонентів, невідомі і недоступні:

```
type
   ObjType=object
                Звичайні поля і методи
             private
                Сховані поля і методи
   end;
```

У TP 6.0 сховані поля і методи описувалися тільки після звичайних полів і методів. У TP 7.0, крім директиви **private**, введена директива **public**. Усі поля і методи, що описані після директиви **public**, доступні усюди, так само як і поля і методи, які описані відразу ж після заголовка об’єктового типу. Розділи **public** і **private** можуть чергуватися довільним чином усередині одного об’єктового типу:

```
type
   TObject=object
                  поля;    {загальнодоступні}
                  методи;  {загальнодоступні}
                private
```

```
                поля;    {сховані}
                методи;  {сховані}
              public
                поля;    {загальнодоступні}
                методи;  {загальнодоступні}
              private
                поля;    {сховані}
                методи;  {сховані}
              public
                поля;    {загальнодоступні}
                методи;  {загальнодоступні}
        end;
```

Якщо в об'єктовому типі, описаному в модулі, яке-небудь поле оголошене схованим, то в об'єкті-спадкоємці можна оголосити поле з тим же ім'ям, але за умови, що спадкоємець описаний у програмі, яка використовує модуль.

8.5. Таблиця віртуальних методів

Якщо об'єктовий тип містить чи успадковує віртуальні методи, конструктори і деструктори, то для нього автоматично створюється таблиця віртуальних методів (ТВМ). У цій таблиці вказується розмір екземплярів об'єктів, що будуть створюватися чи знищуватися з використанням конструкторів і деструкторів, а також посилання на віртуальні методи, доступні для даного об'єктового типу (у тому числі і на успадковані). При цьому для кожного об'єктового типу (але не для кожного екземпляра) створюється своя ТВМ, у котру потрапляють посилання на всі доступні віртуальні методи, навіть якщо відповідні посилання на ті ж самі методи існують у ТВМ предка (у випадку спадкування віртуальних методів).

Конструктор установлює зв'язок між створюваним екземпляром об'єкта і ТВМ відповідного типу, забезпечуючи тим самим доступ екземпляра об'єкта до його віртуальних методів. Тому при звертанні до віртуального методу по його імені завжди викликається той метод, який є характерним для об'єктового типу, що відповідає екземпляру об'єкта. Деструктор при його роботі розриває зв'язок між екземпляром об'єкта і ТВМ.

Якщо екземпляр об'єкта не ініціалізований за допомогою конструктора, то для нього відсутній зв'язок з ТВМ, у результаті чого

робота з таким об’єктом виявляється неможливою при першому ж виклику віртуального методу. Оскільки зв’язок з ТВМ установлюється для конкретного екземпляра об’єкта, робота з неініціалізованим за допомогою конструктора екземпляром об’єкта неможлива навіть у випадку, коли в нього переписана інформація з екземпляра об’єкта, який пройшов ініціалізацію.

У ТР введена функція **TypeOf**, яка повертає покажчик на ТВМ. Ця функція повертає значення типу **Pointer** і має один параметр, що може бути або екземпляром об’єкта, або ім’ям об’єктового типу. Її можна застосовувати тільки до тих об’єктових типів, що мають віртуальні методи (у противному випадку генерується помилка).

Функція **SizeOf** при застосуванні до екземпляра об’єктового типу, який має віртуальні методи, повертає дійсний розмір екземпляра об’єкта (він зчитується з ТВМ), а не той розмір, що визначається описом об’єктового типу.

Якщо предок і спадкоємець описувалися в різних модулях і в предка який-небудь метод оголошений схованим, то в спадкоємця можна оголосити з тим же ім’ям статичний метод.

8.6. Динамічні методи

У ТР, крім віртуальних, уведені так звані динамічні методи, які є підкласом віртуальних методів. Відмінність динамічних методів від віртуальних виявляється в способі їхнього виклику.

Щоб який-небудь метод оголосити динамічним, необхідно усього лише слідом за словом **virtual** вказати ціле додатне число (індекс динамічного методу). Усередині одного об’єктового типу індекси динамічних методів не повинні повторюватися. Діапазон їхніх значень – від 1 до 65535.

Правила перевизначення динамічних методів збігаються з відповідними правилами для віртуальних методів. Додатковою умовою є наступне: перевизначення слідом за словом **virtual** повинне мати той же індекс динамічного методу, що заданий у типі об’єкта-предка для переобумовленого методу. Якщо спадкоємець містить динамічний метод, а предок описувався в іншому модулі і містить оголошений схованим динамічний метод з тим же ім’ям, то однойменні динамічні методи предка і спадкоємця повинні мати різні індекси. Кількість і типи параметрів при цьому непринципові.

У чому перевага динамічних методів перед віртуальними? Як це уже відзначалося, таблиці віртуальних методів створюються для всіх об'єктових типів, які містять віртуальні методи. При цьому, навіть якщо віртуальний метод не перевизначений, а успадкований, він усе рівно потрапляє у ТВМ. Якщо в програмі використовується довгий ланцюжок об'єктів з великою кількістю віртуальних методів, буде витрачатися великий обсяг пам'яті для збереження таблиць віртуальних методів. При використанні динамічних методів разом з ТВМ створюється таблиця динамічних методів (ТДМ). При цьому у ТВМ розміщаються віртуальні методи, а в ТДМ – динамічні, причому успадковані динамічні методи в ТДМ не попадають, а для їхнього пошуку використовується ТДМ об'єкта-предка. Таким чином, застосовуючи динамічні методи, можна домогтися значної економії пам'яті (правда, за рахунок збільшення часу звертання до методів).

Як же взаємодіють ТВМ і ТДМ, і як здійснюється виклик динамічних методів?

Як і раніше, у ТВМ потрапляють усі віртуальні методи. Але якщо об'єктовий тип містить хоча б один динамічний метод, то додатково створюється ТДМ. При цьому у ТВМ заноситься посилання на ТДМ, що дозволяє відшукувати ТДМ об'єкта. У ТДМ потрапляють динамічні методи, визначені чи перевизначені в даному об'єкті. Якщо в об'єкта є предок, для якого створена ТДМ, то в ТДМ об'єкта-спадкоємця міститься посилання на ТДМ об'єкта-предка. Оскільки ТДМ предка досяжна з ТДМ спадкоємця, спадкоємцем може бути викликаний успадкований динамічний метод за допомогою ТДМ предка. При виклику динамічного методу з конкретним індексом через ТВМ йде звертання до його ТДМ. Далі здійснюється пошук методу з даним індексом у всіх ТДМ ланцюжка спадкування з викликом самого останнього у ланцюжку ТДМ динамічного методу.

9. ІНШІ МОЖЛИВОСТІ МОВИ

9.1. Настроювання змінних за адресами

У TP передбачена можливість розміщення змінних за конкретними адресами оперативної пам'яті. Такі змінні називаються абсолютними і розміщуються при описі директивою **absolute**.

Адреси в директиві **absolute** можна задавати двома способами. Перший спосіб складається в накладенні описуваної змінної на змінну, раніше розміщену в пам'яті:

```
type
   string6=string[6];
   TScreenByte=array[1..2000,0..1] of Byte;
   TScreenRec=array[1..2000] of record
                                   ch:Char;
                                   attr:Byte
                                end;
var
   r:Real;
   z:Real absolute r;
   st:string6;
   StLength:Byte absolute st;
   ScrByte:TScreenByte;
   ScrRec:TScreenRec absolute ScrByte;
```

Тут змінна *z* займе ту ж адресу пам'яті, що і *r*; масив ScrRec, у якому зберігаються символи і їхні колірні атрибути, сполучається з масивом ScrByte, у нульовому стовпці якого будуть розміщатися коди символів; змінна StLength накладається на елемент st[0], що дозволяє працювати з довжиною рядка як з числовим значенням (а не символьним, як це має місце при роботі з st[0]).

У загальному випадку типи (а отже, і розміри) змінних, які сполучаються, не збігаються. Розміщення в пам'яті за директивою **absolute** зводиться до простого накладення на перший байт змінної, котра вже розміщена. Саме за рахунок цього коректним є сполучення змінних StLength і st.

Якщо сполучаються змінні складних типів, рекомендується оголошувати типи в розділі **type**, а не конструювати їх по ходу оголошення змінних. У іншому випадку можлива поява помилок, які не виловлюються компілятором.

Програма, що наводиться нижче, ілюструє найпростіший випадок накладення змінних за допомогою директиви **absolute**.

```
{Приклад 9.1}
{Даний рядок з 6 символів. Роздрукувати його 6 разів,
 щораз знищуючи останній символ}
type
   string6=string[6];
var
   st:string6;
   StLength:Byte absolute st;
   i:Byte;
begin
   st:='Pascal';
   for i:=1 to 6 do begin
      WriteLn(st);
      StLength:=StLength-1
   end
end.
```

Щоб при роботі зі змінними, що сполучаються, не псувалися значення інших змінних, потрібно сполучати більш короткі змінні з уже розміщеними більш довгими. Нехай, наприклад, у наведеній програмі розділ **var** має таку структуру:

```
var
   StLength:Byte;
   i:Byte;
   st:string6 absolute StLength;
```

У цьому випадку змінна st накладається не тільки на StLength, але й на *i*, оскільки вони в пам'яті будуть розташовуватися друг за другом. Природно, зміна значення *i* буде відбиватися і на вмісті рядка st (і навпаки).

Другий спосіб настроювання змінних – явна вказівка адреси в директиві **absolute**:

```
type
   TScreen:array[1..4000] of Byte;
var
   Memory:Byte absolute $0000:$0417;
   Screen:TScreen absolute $B800:$0000;
   MemWord:Word absolute $0:$2;
```

У цьому випадку зазначена після **absolute** адреса – це адреса першого байта пам'яті, починаючи з якої розміщується описана змінна.

Адреса задається у вигляді двох констант цілого типу, що не можуть виходити з діапазону $0000..$FFFF (0..65535). Можна також задати адресу шляхом звертання до повертаючих значення типу **Word** функцій **CSeg**, **DSeg** і **SSeg** (відповідно номери сегментів коду, даних і стека). Крім цих функцій, існують ще й інші функції для роботи з адресами змінних:

Addr(ім'я**)** – повертає адресу, за якою розташовується в пам'яті молодший байт змінної чи підпрограми з ім'ям, зазначеним як параметр;

Seg(ім'я**)** – повертає значення типу **Word**, яке визначає номер сегмента, у якому розташовується змінна чи підпрограма з зазначеним ім'ям;

Offs(ім'я**)** – повертає значення типу **Word**, що задає зсув у сегменті адреси молодшого байта змінної чи підпрограми з зазначеним ім'ям;

CSeg – номер сегмента, у якому розташовується код програми (тип **Word**);

DSeg – номер сегмента, у якому розташовуються дані (тип **Word**);

SSeg – номер сегмента, у якому розташовується стек (тип **Word**).

9.2. Робота з буфером екрана

Для підвищення ефективності роботи програм, зокрема для збільшення швидкості виводу на екран, забезпечення створення і швидкої змінюваності декількох сторінок екрана, відновлення потрібного в даний момент екранного зображення, зручно використовувати буфер екрана, що іноді називають картографічною областю пам'яті. Кожній комірці цієї області пам'яті апаратно поставлена у відповідність певна позиція (символ чи точка) екрана. Періодичний перегляд буфера електронними схемами забезпечують його вивід на екран. Відновлення вмісту буфера природно спричиняє зміну зображення на екрані.

Структура й обсяг буфера залежать від типу адаптера. Сама відеопам'ять розташована в адресах пам'яті від $A000:$0000 до $BFFF:$0000. Монохромні режими для текстового зображення використовують комірки пам'яті, починаючи з $B000:$0000, а кольорові – з $B800:$0000.

Рядки екрана зберігаються послідовно, вишиковуючись в ланцюжок від стартової адреси відеопам'яті. Один символ на екрані займає 2 байти пам'яті – перший байт (з парною адресою) містить код символу, другий (непарна адреса) – колірний атрибут.

Організувати доступ до відеопам'яті можна, наклавши на неї регулярну структуру (найчастіше масив). При дозволі 80×25 цей масив повинен займати 4000 байт пам'яті. Структура масиву може бути різною:

```
type
   BufType1=array[1..4000] of Byte;
   BufType2=array[1..2000] of record
                                 ch:Char;
                                 attr:Byte
                              end;
   BufType3=array[1..25,1..80] of record
                                    ch:Char;
                                    attr:Byte
                                 end;
   BufType4=array[1..25,1..80,1..2] of Byte;
```

Якщо відомо поточне число стовпців M на екрані, то перетворення двовимірних координат X та Y у номер k елемента одновимірного масиву провадиться за формулою

$$k=M\cdot(Y-1)+X,$$

якщо елементи масиву двохбайтові. Зворотно: k-му елементу масиву відповідає на екрані стовпець $X=k \bmod M$ і рядок $Y=k \operatorname{div} M+1$.

```
{Приклад 9.2}
{Сформувати 10 випадкових текстових екранних
 зображень і записати у файл}
uses Crt;
type
   t_element=record                        {Тип елементів екрана}
             ch:Char;              {Байт для збереження символу}
             at:Byte; {Байт для збереж. кольорового атрибута}
   end;
   t_scr=array[1..25,1..80] of t_element;          {Тип екрана}
var
   screen:t_scr absolute $B800:0;                        {Екран}
   x,y:Byte;
   k:Word;
   f:file of t_scr;
```

```
begin
   Assign(f,'screens.dat');
   Rewrite(f);
   Randomize;
   for k:=1 to 10 do begin                          {10 зображень}
     for x:=1 to 80 do
      for y:=1 to 25 do begin
       screen[y,x].ch:=Chr(Random(50)+31); {Випадковий символ}
       screen[y,x].at:=k
      end;
     Write(f,screen);                  {Запис зображення у файл}
   end;
   Close(f)
end.
```

Директива **absolute** може бути використана для організації багатосторінкової роботи в текстовому режимі на ТР.

```
{Приклад 9.3}
{Приклад запам'ятовування і відновлення вмісту екрана}
uses Crt;
type
   ScreenType=array[1..4000] of Byte;
var
   Screen:ScreenType absolute $B800:0;                  {Екран}
   Save:ScreenType;     {Масив для збереження сторінки екрана}
begin
   ClrScr;
   GotoXY(20,10);
   Write('0000000');         {Формування текстового зображення}
   Save:=Screen;           {Зберігаємо екран як другу сторінку}
   Delay(5000);
   GotoXY(20,10);
   Write('1111111');                          {Зміна зображення}
   Delay(5000);
   Screen:=Save;                        {Відновлюємо вміст екрана}
end.
```

Для тимчасового збереження екранних зображень найчастіше застосовуються динамічні масиви.

```
{Приклад 9.4}
{Запам'ятати текстовий екран, що був на початку
 роботи програми. Сформувати нове зображення.
 При натисканні будь-якої клавіші в частині екрана по 25-у
 колонку рядки з парними номерами повертаються
 на 180 градусів. При натисканні <Esc> відновлюється
 збережене зображення}
```

```
uses Crt;
type
    TElement=record
               c:Char;
               a:Byte
    end;
    TScreen=array [1..25,1..80] of TElement;
var
     Screen:TScreen absolute $B800:0;
     x,y:Integer;
     Save:^TScreen;
     p:TElement;
     ch:Char;
begin
  New(Save);           {Виділення пам'яті під сторінку екрана}
  Save^:=Screen;                          {Збереження екрана}
  for y:=1 to 25 do             {Формування нового зображення}
    for x:=1 to 80 do
      with Screen[y,x] do begin
        c:=Chr(x);                                      {Символ}
        a:=x                                           {Атрибут}
      end;
  while ch<>#27 do
    if KeyPressed then begin
      ch:=ReadKey;
      if ch=#0 then ch:=ReadKey;
      y:=2;
      while y<25 do begin
        for x:=1 to 25 div 2 do begin  {Поворот частини рядка}
          p:=Screen[y,x];
          Screen[y,x]:=Screen[y,25-x+1];
          Screen[y,25-x+1]:=p
        end;
        y:=y+2
      end
    end;
  Screen:=Save^                          {Відновлення зображення}
end.
```

Як ця, так і програма, що наводиться нижче, демонструють, як можна здійснювати вивід на екран без використання операторів **Write** і **WriteLn**. При цьому можливий вивід навіть у позицію (80,25).

```
{Приклад 9.5}
{У файлі image.dat зберігається текстове зображення
 на повний екран при дозволі 80x25. Організувати
 випадковий прояв цього зображення}
uses Crt;
```

```
type
   t_element=record                          {Тип елементів екрана}
               ch:Char;
               at:Byte;
   end;
   t_scr=array[1..25,1..80] of t_element;          {Тип екрана}
var
   screen:t_scr absolute $B800:0;                          {Екран}
   save:^t_scr;              {Масив для читання екрана з файла}
   x,y:Byte;                                  {Екранні координати}
   k:0..2000;             {Лічильник заповнених позицій екрана}
   f:file of t_scr;
begin
   Assign(f,'image.dat');
   Reset(f);
   New(save);
   Read(f,save^);                         {Читання екрана з файла}
   Randomize;
   k:=0;
   repeat                                     {Цикл виводу зображення}
      x:=Random(80)+1; y:=Random(25)+1;
      if save^[y,x].ch<>#0 then begin
         screen[y,x]:=save^[y,x];     {Вивід символу на екран}
         save^[y,x].ch:=#0;      {Знищення виведеного символу}
         k:=k+1;
         Delay(100);
      end
   until k=1900;          {Поки не буде виведено 95% символів}
   for y:=1 to 25 do                 {Виводимо інші 5% символів}
      for x:=1 to 80 do
        if save^[y,x].ch<>#0 then screen[y,x]:=save^[y,x];
   repeat until KeyPressed;
end.
```

Можна організувати безпосередній обмін даними між файлом і екраном в обох напрямках.

```
{Приклад 9.6}
{У файлі screens1.dat зберігається декілька екранних
 зображень для дозволу 80x25. По натисканню
 стрілок організувати перегляд зображень;
 зображення, що цікавлять, переписати в інший файл}
uses Crt;
type
   t_screen=array[1..4000] of Byte;                {Тип екрана}
var
   screen:t_screen absolute $B800:0;                      {Екран}
   f1,f2:file of t_screen;
   c:Char;
```

```
begin
   Assign(f1,'screens1.dat');
   Reset(f1);
   Assign(f2,'screens2.dat');
   Rewrite(f2);
   ClrScr;
   Write('Стрілки "Угору" і "Униз" - перегляд; ');
   WriteLn('<Enter> - запис; <Esc>- вихід');
   c:=ReadKey;
   while c<>#27 do begin
      case c of
      #0:begin
          c:=ReadKey;
          case c of
          #72:if FilePos(f1)-2<0 then Write(#7)
              else begin
               Seek(f1,FilePos(f1)-2);
               Read(f1,screen);       {Читання екрана з файла}
              end;
          #80:if Eof(f1) then Write(#7)
              else Read(f1,screen);
          end;
         end;
      #13:Write(f2,screen)                {Запис екрана у файл}
      end;
      c:=ReadKey;
   end;
   Close(f2);Close(f1);
end.
```

Природно, як і з будь-яким масивом даних, з відеоекраном можна працювати не тільки як з єдиним цілим, але і з його окремими фрагментами. Наступна програма ілюструє можливість роботи з прямокутними областями текстового екрана.

```
{Приклад 9.7}
{Організувати переписування у файл прямокутного фрагмента
 текстового зображення з координатами (X1,Y1) - лівий верхній
 кут, (X2,Y2) - правий нижній кут. Прочитати спрайт з файла}
uses Crt;
type
   t_element=record                        {Тип елементів екрана}
              ch:Char;
              at:Byte;
   end;
   t_scr=array[1..25,1..80] of t_element;          {Тип екрана}
   t_file=file of t_element;
procedure CopyTextSpriteInFile(x1,y1,x2,y2:Byte;var f:t_file);
```

```
var
   screen:t_scr absolute $B800:0;                          {Екран}
   x,y:Byte;
   r:t_element;
begin
   with r do begin                              {Розміри спрайта...}
      ch:=Chr(y2-y1+1);                      {... по горизонталі...}
      at:=x2-x1+1;                {... і вертикалі заносяться в...}
   end;                                                  {... змінну r}
   Write(f,r);             {Запис розмірів першим елементом файла}
   for y:=y1 to y2 do          {Цикл запису фрагмента зображення}
      for x:=x1 to x2 do Write(f,screen[y,x]);
end;
procedure CopyFileInTextScreen(x1,y1:Byte;var f:t_file);
var
   screen:t_scr absolute $B800:0;                          {Екран}
   x,y:Byte;
   r:t_element;
begin
   Read(f,r);                             {Читання розмірів спрайта}
   for y:=1 to Ord(r.ch) do  {Ord(r.ch) - розмір по вертикалі}
     for x:=1 to r.at do          {r.at - розмір по горизонталі}
        Read(f,screen[y1+y-1,x1+x-1]);      {Спрайт - на екран}
end;
var
   x1,y1,x2,y2:Byte;                             {Координати спрайта}
   f:t_file;
begin
   Assign(f,'sc1');Rewrite(f);
{Продовження програми з формуванням текстового зображення на
 екрані і завданням координат x1,y1,x2,y2 двох кутів спрайта }
   CopyTextSpriteInFile(x1,y1,x2,y2,f);       {Спрайт – у файл}
   Close(f);
{Продовження програми з формуванням нового текстового
 зображення на екрані і завданням координат x1,y1 лівого
 верхнього кута для виводу спрайта}
   Reset(f);
   CopyFileInTextScreen(x1,y1,f);  {Спрайт з файла – на екран}
   Close(f);
end.
```

9.3. Прямий доступ до пам'яті і портів

У TP є три визначені масиви **Mem, MemW, MemL**, елементи яких мають відповідно типи **Byte**, **Word**, **LongInt**. Ці масиви інтерпретують пам'ять як послідовність байтів, слів і подвійних слів і використовуються для прямого доступу до пам'яті.

Як і у випадку звичайних масивів, при звертанні до елементів масивів пам'яті достатньо вказувати індекс. Оскільки в цьому випадку значення індексу задає фізичну адресу елемента пам'яті, то він вказується у вигляді «сегмент:зсув».

Приклади.

`Mem[$B800:0]:='$';Mem[$B800:1]:=Green+16*Red;` – вивід у верхньому лівому куті екрана символу $ зеленим кольором на червоному фоні;

`Mem[$B800:3998]:=36; Mem[$B800:3999]:=Green+16*Red;` – те ж для правого нижнього кута;

`w:=MemW[0:20];` – вміст двох байтів, починаючи з адреси $0000:$0014, записується як число типу **Word** у змінну w;

`LongVar:=MemL[Seg(st):Ofs(st)];` – 4 байти, починаючи з першого байта змінної st, як число типу **LongInt** переписується в змінну LongVar.

Доступ до портів введення/виводу центрального процесора в TP можна організувати за допомогою двох перевизначених одновимірних масивів **Port** і **PortW** з елементами типу **Byte** і **Word** відповідно. Їхні елементи – порти, а індекси (тип **Word**) – адреси портів. Запис значення в елемент масиву **Port** чи **PortW** означає вивід цього значення в зазначений порт. Використання ж якого-небудь елемента одного з цих масивів приводить до введення з відповідного порту. Посилання на який-небудь з цих масивів, як на єдине ціле, заборонені. Крім того, елементи цих масивів не можуть використовуватися як **var**-параметри.

Приклади.

`Port[$20]:=$24;` – вивід у порт із номером $20 значення $24;

`Port[Base]:=Mask xor Port[Base];` – обробляються значення Mask і значення в порту з номером, який зберігається в Base; результат – у порт із номером Base.

9.4. Обробка параметрів командного рядка

Програма, написана на TP, може обробляти невеликий обсяг текстової інформації, переданої їй при запуску. Ця інформація являє собою параметри, що задаються в командному рядку при запуску exe-файла. Параметри відокремлюються один від одного і від імені програми (exe-файла) прогалинами:

D:\user\example.exe param1 param2 param3

Якщо програма запускається з інтегрованого середовища, то параметри задаються в пункті меню OPTIONS/PARAMETRS. Довідатися про кількість параметрів, з якими була запущена програма, можна за допомогою функції без параметрів **ParamCount** (результат має тип **Word**), а звернувшись до функції

ParamStr(номер_параметра**)**,

можна одержати значення параметра (тип **string**) із зазначеним номером (тип **Word**). Параметром з номером 0 є повне ім'я запущеної програми.

```
{Приклад 9.8}
{Написати програму, що поєднує файли, імена яких
 перелічені в параметрах запуску, у файл з ім'ям,
 заданим останнім параметром запуску}
var
   f1,f2:file of Char;
   c:Char;
   i:Integer;
begin
 if ParamCount>1 then begin  {Задані імена хоча б двох файлів}
   Assign(f2,ParamStr(ParamCount));     {Останній параметр...}
   Rewrite(f2);              {...- це ім'я результуючого файла}
   for i:=1 to ParamCount-1 do begin    {i-й параметр - це...}
      Assign(f1,ParamStr(i));             {...ім'я i-го файла}
      Reset(f1);
      while not Eof(f1) do begin
         Read(f1,c);
         Write(f2,c);
      end;
      Close(f1);
   end;
   Close(f2)
 end
 else WriteLn('Невірно зазначені параметри')
end.
```

Програма спочатку визначає кількість параметрів і, якщо їх не менше двох (повинні бути задані імена результуючого і хоча б одного вхідного файла), зчитується останній параметр (з номером **ParamCount**) і відкривається результуючий файл. Далі зчитуються параметри командного рядка від першого до передостаннього (ParamCount-1) і обробляються файли з іменами, що містяться в цих параметрах.

9.5. Безтипові параметри підпрограм

У TP **var**-параметри підпрограм (а в TP 7.0 і **const**-параметри) можуть оголошуватися без вказівки типу (безтипові параметри). Фактичний параметр, що відповідає такому формальному параметру, повинен являти собою змінну будь-якого типу (але не константу чи вираз). Для забезпечення роботи з безтиповим параметром його потрібно погодити в підпрограмі з фактичним параметром відповідно до типу останнього. Це забезпечується або приведенням типу формального параметра до типу відповідного фактичного параметра, або накладенням у пам'яті на безтиповий параметр деякої локальної змінної підпрограми.

Використання безтипових параметрів робить підпрограми більш гнучкими, але від програміста вимагає більшої відповідальності.

```
{Приклад 9.9}
{Функція визначення номера першого елемента, яким
 відрізняються будь-які два одновимірних масиви одного типу з
 однаковою розмірністю. Якщо масиви еквівалентні, то
 результат дорівнює 0}
function Number(var a1,a2;n:Word;SizeElement:Byte):Word;
type
   TMaxByteArray=array[0..MaxInt] of Byte;
var
   i:Word;                              {Номер порівнюваних байтів}
begin
   i:=0;
   while (i<=n*SizeElement)and          {Цикл поки не будуть...}
      {...зіставлені всі байти чи не буде виявлена відмінність}
         (TMaxByteArray(a1)[i]=TMaxByteArray(a2)[i]) do i:=i+1;
   if i>n*SizeElement then Number:=0     {Усі байти збігаються}
   else Number:=i div SizeElement+1      {Обчислення номера...}
end;                           {... елементів, що відрізняються,}
var
   a1,a2:array[1..100] of Integer;
   n,i,p:Word;
begin
   WriteLn('Введіть кількість елементів масивів');ReadLn(n);
   WriteLn('Введіть вміст першого масиву');
   for i:=1 to n do ReadLn(a1[i]);
   WriteLn('Введіть вміст другого масиву');
   for i:=1 to n do ReadLn(a2[i]);
   p:=Number(a1,a2,n,SizeOf(a1[1]));
   if p=0 then WriteLn('Масиви збігаються')
   else WriteLn('Масиви відрізняються ',p,'-м елементом')
end.
```

У наведеній програмі функція містить два безтипові параметри. Їхні типи приводяться до типу TMaxByteArray, що дозволяє працювати з ними як з байтовими масивами, в яких порівнюються відповідні елементи (байти). Параметрами функції є два безтипових параметри (a1 і a2) для передачі порівнюваних масивів, кількість порівнюваних елементів у масивах (n) і розмір кожного елемента в байтах (SizeElement). Значення параметра SizeElement визначається в програмі за допомогою функції **SizeOf.**

Наступна програма дає уявлення про можливу методику роботи з безтиповими параметрами за допомогою накладення на них локальних змінних підпрограми.

```
{Приклад 9.10}
{Процедура зведення в цілу невід'ємну ступінь даних
 типу LongInt і Real. Результат має тип основи
 ступеня. Для інших числових типів обчислення не
 здійснюються. Контроль діапазону не провадиться}
procedure Involution(var x,Result; n:Integer;Size:Byte);
    {Смисл параметрів: x - основа ступеня,
                       Result - результат,
                       n - показник ступеня,
                       Size - кількість байт, що відводиться
                       під основу ступеня (результат)        }
var
   x:LongInt absolute x;             {Якщо x і Result мають...}
   Result: LongInt absolute Result;           {...тип LongInt}
   x:Real absolute x;                {Якщо x і Result мають...}
   Result: Real absolute Result;                 {...тип Real}
   i:Integer;
begin
   case Size of         {Залежно від значення Size працюємо...}
     SizeOf(LongInt):if x=0 then Result:=0       {...з даними}
                     else begin           {...типу LongInt...}
                        ResultL:=1;
                        for i:=1 to n do ResultL:=ResultL*xL
                     end;
     SizeOf(Real):   if x=0 then Result:=0        {...чи Real}
                     else begin
                        ResultR:=1;
                        for i:=1 to n do ResultR:=ResultR*xR
                     end
   end
end;
var
   x,Res1:LongInt;
   r,Res2:Real;
```

```
begin
   x:=3;r:=-1.1;
   Involution(x,Res1,3,SizeOf(LongInt));    {Дані типу LongInt}
   Involution(r,Res2,3,6);                     {Дані типу Real}
   WriteLn(Res1);
   WriteLn(Res2);
end.
```

Тут усередині процедури на безтипові параметри накладені змінні типів **LongInt** та **Real** і залежно від типу фактичних параметрів відбувається розгалуження процесу обчислень, що здійснюються в здійсненній частині процедури.

9.6. Використання асемблерних вставок і інструкцій машинного коду. Вставка фрагментів, написаних на TP

У програмі на TP можливе використання асемблерних вставок. Для цього необхідно розташувати написаний на асемблері фрагмент програми між службовими словами **asm** і **end**:

asm
Фрагмент програми на асемблері
end;

Якщо оператор **asm** заміняє виконавчу частину функції, то значення, що повертається, повинно бути занесено в спеціальну змінну **@Result**.

Якщо вся підпрограма (крім заголовка) написана на асемблері, то слідом за її заголовком (через крапку з комою) вказується службове слово **assembler**, а далі після крапки з комою між службовими словами **asm** і **end** розміщується асемблерний текст підпрограми:

```
procedure ім'я_процедури(опис_форм_параметрів);assembler;
     asm
        Асемблерний текст
     end;
```

У програмах на TP можливе використання підпрограм, цілком написаних на асемблері і записаних після асемблювання в деякий об'єктний файл (стандартне розширення .obj). Для використання такої підпрограми необхідно в описовій частині програми за допомогою директиви компіляції **$L** підключити файл з об'єктним кодом підпрограми, після чого звичайним чином оголошується заголовок підпрограми з вказівкою (через крапку з комою) службового слова **external**:

```
uses Crt;
{$L lib.obj}                        {Підключення файла lib.obj}
function Hypoten(a,c:Real):Real;external;  {Оголошення...}
                          {...заголовка зовнішньої підпрограми}
begin
        {Продовження програми}
end.
```

Крім асемблерних вставок, у ТР передбачена можливість використання інструкцій машинного коду, для чого служить **inline**-оператор, що складається зі службового слова **inline**, за яким у круглих дужках йдуть елементи машинного коду, розділені косою рискою.

У ТР процедури і функції можуть компілюватися з директивою **inline**. Подібні підпрограми описуються своїм заголовком, слідом за яким розміщується конструкція, що збігається за написанням з **inline**-оператором.

В описовій частині програми дозволено вставляти тексти, що написані на ТР і зберігаються в текстових файлах. Для цього у відповідному місці програми вказують директиву компіляції **$I**, у якій як параметр використовують ім’я текстового файла (наприклад, {$I f.pas}). Компіляція програми в цьому випадку провадиться так, ніби вміст файла, що включається, знаходиться в програмі безпосередньо в місці розташування директиви **$I**. Подібним чином в текст програми можуть бути включені підготовлені у вигляді окремих файлів константні масиви, процедури, функції і т.п.

10. МОДУЛЬ DOS

10.1. Перевірка результату звертання до засобів MS DOS

У модулі **Dos** зосереджені функції і процедури, що забезпечують доступ до засобів MS DOS. При звертанні до засобів операційної системи, зокрема при використанні деяких процедур модуля **Dos,** можлива поява помилок. Для їхнього виявлення введена змінна **DosError**, в яку заноситься код помилки. Можливі значення цієї змінної такі:

0 – нормальне завершення;
2 – файл не знайдений;
3 – шлях не знайдений;
4 – занадто багато відкритих файлів;
5 – доступ до файла закритий;
6 – порушена інформація в полях файла чи системних областях;
8 – недостатньо пам'яті;
10 – несумісність параметрів оточення;
11 – нерозпізнаваний формат диска;
18 – немає більше файлів.

Рекомендується здійснювати контроль коректності виконання процедур модуля **Dos** за допомогою звертання до змінної **DosError** відразу ж слідом за викликом процедур, що можуть генерувати наведені вище коди повернення. Це можна зробити, наприклад, в операторі **if**:

```
if DosError<>0 then begin
  WriteLn('Виникла помилка Dos з номером ',DosError);Halt
end;
```

10.2. Запуск зовнішніх програм і внутрішніх команд DOS

Для запуску прикладної програми з програми на TP служить процедура

Exec(команда, параметр**).**

Її перший параметр (тип **PathStr**) – це ім'я файла, що містить прикладну програму (диск, шлях, ім'я, розширення); другий параметр

(тип **string**) – рядок, що містить параметри відповідного командного рядка MS DOS. Тип PathStr=string[79] – стандартний тип модуля **Dos**.

Якщо зовнішня програма, що викликається за допомогою **Exec**, не вимагає параметрів (наприклад, у програмі, яка наводиться нижче, просто викликається редактор), то другий параметр процедури **Exec** має вигляд порожнього рядка **''**. Треба пам'ятати, що для виконання програми, що викликається, необхідна вільна пам'ять. Тому потрібно акуратно відноситися до керування динамічною пам'яттю за допомогою директиви **$M**, а саме, у цьому випадку програма, яка здійснює виклик, не повинна займати всю динамічну пам'ять.

Запуск внутрішніх команд MS DOS здійснюється за допомогою звертання до командного процесора command.com з ключем /C:

```
Exec('C:\command.com', '/C dir')
```

Бажано, щоб програм, що викликається, виконувалася в оточенні, яке притаманне операційній системі. Програма на TP звичайно перевизначає деякі переривання. Тому перед виконанням процедури **Exec** рекомендується відновити оточення DOS, а після виконання викликаної програми повернутися до старого оточення. Переключення стану оточення здійснює процедура без параметрів

SwapVectors,

яку рекомендується викликати безпосередньо до і після звертання до **Exec**.

Якщо команда MS DOS, що викликається, чи прикладна програма не можуть бути виконані, то повідомлення про помилку не видається і головна програма продовжує виконуватися. Тому після виклику **Exec** бажана перевірка значення змінної **DosError**.

```
{Приклад 10.1}
{Для обробки файла Message.txt, який знаходиться у поточному
 каталозі, викликати з програми на TP редактор q~.exe,
 розташований на диску C: у каталозі TOOLS}
{$M 16384,0,0}                    {Керування пам'яттю для Exec}
uses Dos;
   {. . .}
begin
   {Текст програми}
   SwapVectors;                    {Відновлення переривань DOS}
   Exec('C:\TOOLS\q~.exe','message.txt');
   SwapVectors;          {Повернення стану середовища програми}
   if DosError<>0 then begin
      WriteLn('Виникла помилка DOS з номером ',DosError);Halt
```

```
   end;
  {Продовження програми}
end.
```

У модулі **Dos**, крім можливості контролю правильності виконання його процедур, передбачений механізм перевірки того, як завершилася викликана зовнішня програма. Для цього використовується функція без параметрів

DosExitCode,

результатом якої є значення типу **Word**: молодший байт – код повернення із зовнішньої програми, старший – ознака того, як завершилася програма. Як код повернення виступає або код помилки, або значення параметра процедури **Halt** при завершенні за **Halt**, або 0 при нормальному завершенні. Ознаки способу завершення зовнішньої програми:

0 – нормальне; 1 – по натисканню Ctrl/Break;
2 – аварійне; 3 – процедурою **Keep**.

Нагадаємо, що для зчитування значення в молодшому і старшому байтах використовуються функції **Lo** і **Hi** відповідно:

```
WriteLn('Код повернення з зовнішньої програми ',Lo(DosExitCode));
```

або

```
if Hi(DosExitCode)=1 then
 WriteLn('Зовнішня програма завершилася натисканням Ctrl/Break');
```

10.3. Робота зі змінними оточення

Операційна система відводить область пам'яті для збереження певної системної інформації. Ця інформація зберігається у вигляді текстових рядків зі структурою «ім'я=значення» у так званих змінних оточення. Наприклад, у змінній COMSPEC зберігається повне ім'я командного процесора, а в змінній PATH перелічують (через крапку з комою) шляхи доступу до здійснимих файлів (з розширеннями .com, .exe, .bat), що будуть викликатися не по повному, а по скороченому імені. Значення змінної COMSPEC звичайно задається операційною системою, а в змінну PATH значення записується за допомогою файла AUTUEXEC.BAT:

```
COMSPEC=C:\COMMAND.COM
PATH=C:\DOS;C:\WINDOWS;C:\NU;C:\NC
```

Можна визначити інші змінні оточення.

Для роботи зі змінними оточення в модулі **Dos** існує три функції:

EnvCount – повертає значення типу **Integer**, що дорівнює кількості рядків у середовищі оточення (кількість змінних оточення);

EnvStr(номер**)** – повертає із середовища оточення рядок (тип **string**) із зазначеним номером, що задається значенням типу **Integer** (вигляд результату: *ім'я_змінної_оточення=значення*);

GetEnv(рядок**)** – повертає рядок (тип **string**), що містить значення змінної оточення, ім'я якої задає параметр функції (тип **string**).

```
{Приклад 10.2}
{Які змінні оточення встановлені в системі?
 Роздрукувати всі рядки середовища оточення і видати
 вміст поточного каталога}
{$M 10000,0,0}                    {Керування пам'яттю для Exec}
uses Dos;
var
   i:Integer;
begin
   for i:=1 to EnvCount do           {Цикл по рядках оточення}
      WriteLn(Copy(EnvStr(i),1,Pos('=',EnvStr(i))-1));
   WriteLn('Для продовження натисни <Enter>');ReadLn;
   for i:=1 to EnvCount do
      WriteLn(EnvStr(i));                {Друк рядка оточення}
   WriteLn('Для продовження натисни <Enter>');ReadLn;
   SwapVectors;
   Exec(GetEnv('Comspec'),'/C dir')  ;{Тут  GetEnv зчитує...}
   SwapVectors;          {...повне ім'я командного процесора}
   if DosError<>0 then WriteLn(DosError);
   {Продовження програми}
end.
```

У програмі показано, як через виклик командного процесора виконати внутрішню команду MS DOS (у даному випадку DIR).

10.4. Процедури для роботи з таймером

Процедура

GetTime(години**,** хвилини**,** секунди**,** соті**)**

у своїх параметрах (тип **Word**) повертає поточний час, встановлений у системі (години, хвилини, секунди, соті частки секунди).

Процедура

GetDate(рік**,** місяць**,** число**,** день_тижня**)**

з параметрами типу **Word** повертає відповідно поточний рік (у діапазоні 1980..2099), місяць (у діапазоні 1..12), число (у діапазоні 1..31) і день тижня (у діапазоні 0..6 з відліком від неділі).

```
{Приклад 10.3}
{Визначити час рішення задачі}
uses Dos;
var
   h,m,s,ss,h1,m1,s1,ss1:Word;
begin
   GetTime(h,m,s,ss);
   WriteLn('Скільки буде двічі два?');ReadLn;
   GetTime(h1,m1,s1,ss1);
   WriteLn('Ви думали ',(m1-m)*60+s1-s,' секунд');
end.
```

Аналогічно процедури

SetTime(години**,** хвилини**,** секунди**,** соті**)**

і

SetDate(рік**,** місяць**,** число**)**

з параметрами типу **Word** встановлюють в операційній системі відповідно поточний час (години, хвилини, секунди, соті частки секунди) і дату (рік, місяць, число).

```
{Приклад 10.4}
{Установити дату}
uses Dos;
var
  y,m,d:Word;
begin
  WriteLn('Введіть число, місяць, рік трьома цілими числами');
  ReadLn(d,m,y);
  SetDate(y,m,d)
end.
```

10.5. Перевірка станів

Для перевірки стану диска в модулі **Dos** введені дві функції

DiskSize(номер_диска**);**
DiskFree(номер_диска**),**

які у вигляді значення типу **LongInt** повертають відповідно розмір у байтах і число вільних байтів на диску з зазначеним номером, що задається значенням типу **Byte** (0 – поточний диск, 1 – диск A, 2 – диск B і т.д.).

Процедура

GetCBreak(ВклВикл**)**

з параметром типу **Boolean** визначає, чи активізована перевірка стану клавіші Ctrl/Break (значення, що повертається**: True** – так**, False** – ні), а процедура

SetCBreak(ВклВикл**),**

що також має параметр типу **Boolean**, установлює необхідність перевірки на натискання Ctrl/Break при кожному системному звертанні (значення параметра **True**) або скасовує перевірку (якщо задане значення параметра **False**).

При роботі MS DOS можливі два стани, зв'язані з перевіркою правильності копіювання файлів. Ці стани встановлюються командою VERIFY: VERIFY ON – перевірка включена; VERIFY OFF – виключена.

Модуль **Dos** дозволяє визначити зафіксований у даний момент стан прапора перевірки, для чого служить процедура

GetVerify(ВклВикл**)**

з параметром типу **Boolean**. Вона повертає значення **True** при стані VERIFY ON і **False** при VERIFY OFF.

Процедура

SetVerify(ВклВикл**)**

встановлює в потрібний стан прапор перевірки через значення свого параметра, що має булевський тип (**True** – включити, **False** – виключити).

Щоб довідатися, яка версія MS DOS установлена на комп'ютері, потрібно звернутися до функції без параметрів **DosVersion**, що повертає у виді значення типу **Word** номер версії операційної системи (молодший байт – ціла частина номера версії, старший – дробова).

```
{Приклад 10.5}
{Визначити номер версії MS DOS}
uses Dos;
begin
   WriteLn('Установлена MS DOS версії ',
           Lo(DosVersion),'.',Hi(DosVersion))
end.
```

Процедури перевірки станів у найпростіших програмах практично не використовуються.

10.6. Робота з файлами і каталогами

Функція

FSearch(шлях**,** каталоги**)**

шукає файл, ім'я якого задано в першому параметрі (тип **PathStr**). Другий параметр (тип **string**) – це перелік каталогів, у яких шукається файл (через крапку з комою). Пошук завжди починається з поточного каталогу. Функція повертає у вигляді значення типу **PathStr** повне ім'я файла (диск, шлях, ім'я, розширення). Якщо файл не знайдений, повертається порожній рядок .

```
{Приклад 10.6}
{Якщо на шляхах, встановлених у MS DOS, знаходиться
 програма SpeedDisk (sd.exe) здійснити її запуск.
 У протилежному випадку видати повідомлення}
{$M 10000,0,0}
uses Dos;
var
   s:PathStr;
begin
   s:=FSearch('sd.exe',GetEnv('PATH'));
   if s='' then WriteLn('Програма SpeedDisk не знайдена')
   else begin
      SwapVectors;
      Exec(s,'');
      SwapVectors;
      if DosError<>0 then WriteLn('Помилка ',DosError)
   end;
   {Продовження програми}
end.
```

У програмі за допомогою функції **GetEnv** зчитується вміст змінної оточення PATH, в якій у MS DOS зберігаються шляхи доступу до здійсненних файлів, що викликаються без вказівки повного шляху.

Процедура

FindFirst(шлях**,** атрибути**,** перший_запис**)**

здійснює пошук першого запису, що відповідає заданим імені (перший параметр, тип **string**) і набору атрибутів (другий параметр, тип **Byte**). Запис може відшукуватися не тільки по конкретному імені,

але і серед файлів, що задаються по масці в першому параметрі. Набір атрибутів – це деяке числове значення. Окремим атрибутам відповідають такі константи модуля **Dos**:

ReadOnly=$01,
Hidden=$02,
SysFile=$04,
VolumeID=$08,
Directory=$10,
Archive=$20,
AnyFile=$3F.

Комбінації атрибутів задаються іншими значеннями. Результати роботи процедури повертаються в третьому параметрі, тип якого описаний у модулі **Dos** так:

```
type
    SearchRec=record
       Fill:array[1..21] of Byte;
       Attr:Byte;
       Time:LongInt;
       Size:LongInt;
       Name:string[12]
    end;
```

У цьому параметрі наводяться характеристики знайденого файла: значення атрибута (поле **Attr**), упакований час створення (поле **Time**), розмір файла в байтах (поле **Size**), ім'я і розширення файла або каталогу (поле **Name**). Поле **Fill** не повинне використовуватися, оскільки воно резервується для роботи операційної системи.

Процедура

FindNext(наступний_запис**)**

служить для пошуку на диску наступного запису з тими ж параметрами, що й у раніше виконаній процедурі **FindFirst** чи **FindNext**. Її параметр має тип **SearchRec**.

Якщо пошук за допомогою **FindFirst** чи **FindNext** виявився безуспішним, то в змінну **DosError** автоматично заноситься значення 18 (процедура **FindFirst** може виробляти також код помилки 3).

Упакований час поєднує в собі дату і час останньої корекції файла. Для розпакування цієї інформації служить процедура

UnPackTime(упакований_час**,** розпакований_час**),**

першим параметром (тип **LongInt**) якої є упакований час, а другим – інформація про дату і час у вигляді запису типу **DateTime**, що у модулі **Dos** визначений у такий спосіб:

```
type
     DateTime=record
              Year,Month,Day,Hour,Min,Sec:Word
     end;
```

Процедура

PackTime(розпакований_час**,** упакований_час**)**

упаковує запис про дату і час, що має тип **DateTime**.

Процедури

SetFAttr(файлова_змінна**,** атрибут**)**

і

GetFAttr(файлова_змінна**,** атрибут**)**

служать відповідно для установлення і сприйняття значення атрибута (другий параметр типу **Byte**) файла, що задається першим параметром. Процедури можуть виробляти код помилки 3 чи 5.

Процедура

GetFTime(файлова_змінна**,** упакований_час**)**

повертає упакований час останньої корекції файла, заданого першим параметром, а процедура

SetFTime(файлова_змінна**,** упакований_час**)**

встановлює для файла час останньої корекції. Для установки нового часу створення файла за допомогою процедури **SetFTime** потрібно здійснити попереднє відкриття файла, у той час як при установці атрибутів файл повинен бути закритий. Процедура **SetFTime** може виробляти помилку 6.

```
{Приклад 10.7}
{Якщо в поточному каталозі існує файл test.dat, встановити
 час його створення на 2 хв після 0 год 01.01.2003.
 У поточному каталозі в усіх файлах з розширенням .txt,
 які не мають атрибут "тільки для читання", установити
 згаданий час і додати атрибут прихованості}
uses Dos;
var
   dt:DateTime;
   at:Word;
   file_time:LongInt;
   file_rec:SearchRec;
   f:file;
begin
   with dt do begin              {Завдання встановлюваних ...}
      Year:=2002;Month:=1;Day:=1;                {... дати...}
      Hour:=0;Min:=2;Sec:=0                       {... і часу}
   end;
```

```
   PackTime(dt,file_time);                    {Упакування часу}
   FindFirst('test.dat',AnyFile,file_rec);
   if DosError=0 then begin          {Якщо файл знайдений, ...}
      Assign(f,file_rec.Name);
      Reset(f);                   {...то відкриваємо його і...}
      SetFTime(f,file_time);              {...встановлюємо час}
      Close(f);                               {Закриваємо файл}
   end;
   FindFirst('*.txt',AnyFile,file_rec);      {Пошук першого...}
                               {...файла з розширенням .txt}
   while DosError=0 do begin              {Поки пошук успішний}
      Assign(f,file_rec.Name);
      GetFAttr(f,at);                    {Зчитуємо атрибут файла}
      if at and ReadOnly=0 then          {Якщо немає захисту ...}
      begin           {...від запису, то змінюємо атрибути...}
        SetFAttr(f,at or Hidden);      {...(файл закритий),...}
        Reset(f);           {...відкриваємо файл і змінюємо...}
        SetFTime(f,file_time);          {...час його створення}
        Close(f);
      end;
      FindNext(file_rec)               {Шукаємо наступний файл}
   end
end.
```

У програмі показано, як здійснювати пошук файла з конкретним ім'ям (test.dat) і по масці (*.txt).

Функція

FExpand(шлях**)**

з параметром типу **PathStr** повертає повне ім'я файла (диск, шлях, ім'я, розширення) у вигляді рядка типу **PathStr**.

Перевірка на наявність файла не провадиться: до зазначеного імені йде формальне приєднання шляху до поточного каталога. Якщо заданий як параметр шлях починається із символу «\», то припускається, що він відлічується від кореневого каталогу і відбувається приєднання тільки імені диска. Вказівка в шляху імені надкаталогу (..) формує шлях так, ніби файл розташовувався не в поточному каталозі, а на рівень вище.

```
{Приклад 10.8}
{Робота функції FExpand для поточного каталогу C:\USER\MY_DIR}
uses Dos;
begin                          {Видаються відповідно рядки...}
   WriteLn(FExpand('t.dat'));           {C:\USER\MY_DIR\T.DAT}
   WriteLn(FExpand('..\t.dat'));               {C:\USER\T.DAT}
   WriteLn(FExpand('\US\D1\t.dat'))          {C:\US\D1\T.DAT}
end.
```

Процедура

FSplit(шлях, каталог**,** ім’я**,** розширення**)**

для зазначеного в першому параметрі імені файла (тип **PathStr**) здійснює поділ на три частини – шлях зі зворотною косою наприкінці (другий параметр, тип **DirStr**); ім’я файла (третій параметр, тип **NameStr**), крапка і розширення (четвертий параметр, тип **ExtStr**), якщо вони є в першому параметрі. Використовуються описані в модулі **Dos** типи:

```
DirStr=string[67],
NameStr=string[8],
ExtStr=string[4].
```

Процедура не робить пошук файла і формування повного імені, а просто здійснює розбивку на складові частини рядка, зазначеного як перший параметр.

```
{Приклад 10.9}
{Знайти по шляхах, що встановлені в операційній системі,
 файл sd.exe і одержати складові його повного імені}
uses Dos;
var
   p:PathStr;
   dir:DirStr;
   name:NameStr;
   ext:ExtStr;
begin
   p:=FSearch('sd.exe',GetEnv('PATH'));
   if p='' then WriteLn('sd.exe не знайдений по шляхах')
   else begin
      FSplit(p,dir,name,ext);
      WriteLn(dir);{Диск і шлях зі зворотною косою наприкінці}
      WriteLn(name);                                    {Ім'я}
      WriteLn(ext)                          {Крапка і розширення}
   end
end.
```

10.7. Робота з перериваннями

Дуже часто для одержання більш ефективної програми доводиться прямо звертатися до операційної системи. Для цього в модулі **Dos** є дві процедури: **Intr** та **MsDos**.

Перша процедура дозволяє викликати програмні переривання, через які реалізується доступ до ресурсів операційної системи. Звертання до цієї процедури виробляється в такий спосіб:

Intr(номер_переривання**,** регістри**).**

Перший параметр (тип **Byte**) – це номер програмного переривання (число з інтервалу 0..255); другий параметр – запис, за допомогою якого передаються значення регістрів для зазначеного переривання. Другий параметр повинен бути оголошений з типом **Registers**, який описаний у модулі **Dos** так:

```
type
  Registers=record
   case Integer of
   0:(AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags:Word);
   1:(AL,AH,BL,BH,CL,CH,DL,DH:Byte)
  end;
```

Структура цього запису моделює набір регістрів центрального процесора (варіанти цього запису відбивають той факт, що з першими чотирма регістрами можна працювати або як з єдиними словами, або з їх молодшими і старшими байтами окремо).

При виконанні процедури **Intr** значення полів **AX**, **BX**, **CX**, **DX**, **BP**, **SI**, **DI**, **DS**, **ES** запису-параметра завантажуються в однойменні регістри центрального процесора, слідом за чим здійснюється виклик переривання з зазначеним номером. Після виконання переривання новий вміст цих регістрів заноситься в другий параметр у ті ж поля, а вміст регістра прапорів заноситься в поле **Flags**.

Процедура **Intr** не може використовуватися для виклику тих переривань, що вимагають завдання певних значень регістрів **SP** чи **SS** або змінюють значення цих регістрів.

```
{Приклад 10.10}
{Зчитування поточного часу}
program GetTime;
uses Dos;
type
  TimeStr=string[8];
function Time:TimeStr;
var
  Regs:Registers;
  Hour,Min,Sec:string[2];
begin
  Regs.AH:=$2C;          { $2C - номер функції MS DOS GetTime}
```

```
  Intr($21,Regs);         { $21 - переривання "функції MS DOS"}
  with Regs do
    begin           {Для функції $2C результат у регістрах:...}
        Str(CH,Hour);                    {... CH - години, ...}
        Str(CL,Min);                   {... CL - хвилини, ...}
        Str(DH,Sec);                         {... DH - секунди}
    end;
  Time:=Hour+':'+Min+':'+Sec
end;
begin
  WriteLn(Time)
end.
```

Перед виконанням процедури **Intr** у старшому байті регістра **AX** треба розмістити код $2C, що забезпечує з появою переривання $21 запис у регістри **CX** і **DX** поточного часу: у старших бітах **CX** – години, у молодших – хвилини, у старших бітах регістра **DX** – секунди.

Переривання з номером $21 – «функції MS DOS» – виконує велику кількість дій. Для звертання до нього, крім процедури **Intr**, служить процедура

MsDos(регістри**),**

параметр якої аналогічний другому параметру процедури **Intr**. Результат звертання до процедури **MsDos** той же самий, що і при звертанні до процедури **Intr** з номером переривання $21.

```
{Приклад 10.11}
{Встановити таймер}
uses Dos;
var
  regs:Registers;
begin
  regs.AH:=$2D;                  {Завантаження номера SetTime}
  regs.CH:=12;                                {Установка годин}
  regs.CL:=43;                              {Установка хвилин}
  regs.DH:=00;                              {Установка секунд}
  MsDos(regs);                            {Звертання до SetTime}
  WriteLn(regs.AL)              {0 - нормальне завершення ...}
end.                      {...чи $FF, якщо час заданий невірно}
```

Перед виконанням процедури **MsDos** у регістр **AH** заноситься значення $2D – номер функції установки таймера, а в регістри **CH**, **CL**, **DH** у вигляді цілих значень заносяться відповідно години, хвилини і секунди. Після виконання процедури **MsDos** у регістрі **AL**

повертається значення 0 при нормальній установці часу і значення $FF, якщо час заданий невірно.

Як видно з прикладів, процедура **MsDos** працює аналогічно **Intr**, але для неї не потрібно вказувати номер переривання.

10.8. Перевизначення переривань. Резидентні програми

Дуже часто виникає необхідність визначення власної реакції на переривання операційної системи замість стандартних реакцій. Для цього описуються процедури спеціального виду

IntXX(**Flag,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP:Word**);**interrupt**;

При оголошенні такої процедури після заголовка обов'язково потрібно вказувати службове слово **interrupt** із крапкою з комою. У подібні процедури як псевдопараметри передаються всі регістри, що дозволяє використовувати їх і змінювати. При використанні не всіх регістрів список параметрів можна скорочувати зліва. Компіляція процедур обробки переривань повинна здійснюватися тільки при активізації далекої моделі виклику (директива **{$F+}**).

Процедура обробки переривання може змінювати свої параметри, за рахунок чого будуть модифікуватися відповідні регістри при поверненні з оброблювача переривань.

Для перевизначення переривання в модулі **Dos** введена процедура з двома параметрами

SetIntVec(номер_переривання**,** адреса_процедури**),**

перший параметр якої – номер переривання (тип **Byte**), а другий – покажчик на процедуру, що буде виконуватися з появою переривання з даним номером (наприклад, @IntXX). Якщо через якийсь час потрібно відновити стандартну реакцію на переривання, то перед перевизначенням треба запам'ятати адресу реакції на переривання. Для цього служить процедура

GetIntVec(номер_переривання**,** покажчик**),**

яка заносить у другий параметр (тип **Pointer**) адресу поточного переривання з номером, що задається першим параметром (тип **Byte**). По закінченні роботи з власною процедурою обробки переривання можна відновити реакцію на переривання за допомогою процедури **SetIntVec**.

```
{Приклад 10.12}
{Деякий імітатор вірусу, що активізується по перериванню
 1B (але не сам вірус). Запускати як exe-файл}
uses Dos;
var
  Int1BSave:Pointer;
{F+}                                    {Включаємо далекий виклик}
procedure Int1B;interrupt;                 {Процедура, що буде...}
                    {... викликатися за натисканням Ctrl/Break}
begin
  WriteLn('Formating on driver C:');
end;
{$F-}                                  {Відключаємо далекий виклик}
begin
                 {Запам'ятовуємо адресу стандартної реакції...}
  GetIntVec($1B,Int1BSave);                    {... у Int1BSave}
                            {Замість неї встановлюємо адресу...}
  SetIntVec($1B,@Int1B);                    {... своєї процедури}
  WriteLn('Press Ctrl/Break');
  repeat until False;
  SetIntVec($1B,Int1BSave);                      {Відновлюємо...}
end.                    {...стандартну реакцію (тут не працює) }
```

У процедурах обробки апаратних переривань не рекомендується використовувати будь-які програми введення/виводу, програми динамічного розподілу пам'яті і функції DOS.

Існують так звані резидентні програми. Вони активізуються з появою переривання, на яке вони налагоджені (наприклад, при натисканні потрібної комбінації клавіш). Щоб зробити програму резидентною, необхідно завершити її виконанням процедури

Keep(код_повернення**)**

з одним параметром типу **Word**.

Написання резидентних програм нами розглядатися не буде.

11. МОДУЛЬ GRAPH

11.1. Графічні режими і їх ініціалізація

Перші дисплеї (монітори) були монохромними. Їх представниками були монітори MDA (Mono Digitals Adapter) і Hercules, що зустрічаються зараз надзвичайно рідко. З кольорових моніторів раніше були поширені монітори CGA (Color Graphics Adapter) і EGA (Enhanced Graphics Adapter), зараз VGA (Video Graphics Adapter) і SVGA (Super-VGA). Кольорове зображення виходить за рахунок підсвічування люмінофорних точок трьох кольорів (червоний, синій, зелений). При цьому кожна точка зображення (піксель) утворюється трьома розташованими поруч люмінофорними точками.

Кількість пікселів по горизонталі і вертикалі називається розрізнювальною здатністю монітора. Характеристики моніторів різних типів у графічному режимі такі:

MDA – 640×200, 2 кольори;
CGA – 640×200, 2 кольори чи 320×200, 4 кольори;
Hercules – 720×348, 2 кольори;
EGA – 640×350, 16 кольорів чи 640×200, 16 кольорів;
VGA – 640×480, 16 кольорів.

Монітори EGA і VGA фактично стали стандартом для тих застосувань, що забезпечуються графічними можливостями мови TP. Особливість моніторів VGA у тому, що в них суттєво більша розрізнювальна здатність, причому відстані між сусідніми пікселями по горизонталі і вертикалі збігаються. Взагалі говорячи, монітори EGA дозволяють одержати 64 кольори, а VGA – 256. У моніторах SVGA активними є 256 кольорів .

У системному модулі TP не передбачені можливості по роботі в графічному режимі. Всі засоби подібного роду зосереджені в модулі **Graph**. Тому за допомогою оголошення uses Graph насамперед слід підключити цей модуль. Однак цього недостатньо: графіка повинна бути ініціалізована. Ініціалізація графічної системи забезпечує переведення апаратури в графічний режим. Вона здійснюється процедурою

InitGraph (драйвер**,** режим**,** шлях**).**

Значення першого параметра цієї процедури (тип **Integer**) вказує номер графічного адаптера (драйвера), а другого (тип **Integer**) – номер графічного режиму; третій параметр (тип **string**) – це каталог файла драйвера (для моніторів EGA, VGA і SVGA використовується графічний драйвер egavga.bgi). Якщо третій параметр пустий, то передбачається, що файл драйвера знаходиться в поточному каталозі.

Для першого параметра процедури **InitGraph** в модулі **Graph** визначені константи; деякі з них такі:

Detect=0 – автоматичне визначення типу графічного адаптера;
CGA=1 – адаптер CGA;
EGA=3 – адаптер EGA;
HercMono=7 – адаптер Hercules;
VGA=9 – адаптер VGA.

Аналогічно деякими з констант, що встановлюють графічні режими (значення другого параметра), є:

- для адаптера EGA:
  EGALo=0 – перша палітра, 640×200, 16 кольорів;
  EGAHi=1 – друга палітра, 640×350, 16 кольорів;
  EGAMonoHi=3 – 640×350, 2 кольори;
- для адаптера Hercules:
  HercMonoHi=0 – монохром 720×348, 2 сторінки;
- для адаптера VGA:
  VGALo=0 – 640×200;
  VGAMed=1 – 640×350;
  VGAHi=2 – 640×480.

Для різних графічних драйверів одне й те ж значення другого параметра встановлює режим, який відповідає типу драйвера, заданому першим параметром процедури **InitGraph**.

Якщо перший параметр відмінний від нуля, то його значення розглядається як номер драйвера, і система переводиться в режим, визначений значенням другого параметра.

Якщо перший параметр дорівнює **Detect**, то можна задати будь-яке значення змінній, що підставляється як другий параметр. У цьому випадку автоматично виконується виклик процедури

DetectGraph(драйвер**,** режим**),**

яка визначає тип драйвера, що відповідає монітору, і записує в перший параметр значення, що дорівнює номеру драйвера, а в другий – значення, яке дорівнює максимальному можливому номеру режиму

для даного графічного драйвера. Процедура **DetectGraph** може викликатися і поза процедурою **InitGraph**.

Відзначимо, що параметри процедури **DetectGraph**, як і перші два параметри процедури **InitGraph**, є **var**-параметрами, тому при виклику замість них потрібно підставляти змінні.

Правильність виконання процедури **InitGraph** можна проконтролювати, звернувшись до функції без параметрів **GraphResult**, яка поверне одне зі значень:

grOk=0 – немає помилки;
grNoInitGraph=-1 – графіка не ініціювалась;
grNotDetected=-2 – графічна карта не виявлена;
grFileNotFound=-3 – не знайдений файл драйвера;
grInvalidDriver=-4 – невідповідний файл драйвера;
grNoLoadMem=-5 – нема пам'яті для завантаження драйвера;
grInvalidMode=-10 – неправильний графічний режим.

При повторному звертанні до функції **GraphResult** без попереднього виконання операцій, що можуть змінити її значення, повертається 0.

Для виходу з графічного режиму служить процедура без параметрів **CloseGraph**, яка вивантажує графічний драйвер і відновлює текстовий режим. Зворотний перехід у графічний режим після цього можливий тільки через процедуру **InitGraph**.

```
{Приклад 11.1}
{Ініціалізація і закриття графіки}
uses Graph;
var
   d,m:Integer;
   error:Integer;
begin
   d:=Detect;
   InitGraph(d,m,'C:\TP7\BGI');
   error:=GraphResult;
   if error<>0 then begin
      WriteLn('Помилка ініціалізації графіки номер ',error);
      Halt
   end;
   {Робота в графічному режимі}
   CloseGraph;
   {Робота в текстовому режимі}
end.
```

Для одержання текстового повідомлення про помилку (англійською мовою) можна звернутися до функції

GraphErrorMsg(код_помилки**),**

що повертає значення типу **string** з повідомленням про помилку з заданим кодом (тип **Integer**).

Щоб взнати тип драйвера, можна скористатися функцією без параметрів

GetDriverName,

що повертає значення типу **string** з ім'ям драйвера. Функції без параметрів

GetGraphMode

і

GetMaxMode

у вигляді значень типу **Integer** повертають відповідно номер поточного режиму і максимальний номер режиму для встановленого драйвера. Функція ж

GetModeName(режим**)**

для заданого графічного режиму (тип **Integer**) повертає у вигляді рядка типу **string** дозвіл екрана й ім'я режиму роботи. Діапазон можливих режимів роботи заданого графічного адаптера можна взнати, звернувшись до процедури

GetModeRange(драйвер**,** min_режим**,** max_режим**)**

з параметрами типу **Integer**, які мають таке розуміння: перший параметр – номер драйвера, другий і третій – максимальне і мінімальне значення графічних режимів. При неправильному значенні першого параметра другий і третій параметр одержують значення –1. У цій процедурі перший параметр можна задати константою **CurrentDriver** (значення –128), яка відповідає номеру активного в даний момент драйвера.

```
{Приклад 11.2}
{Функції і процедури перевірки режимів}
uses Graph;
var
   d,m:Integer;
   error:Integer;
   l,h:Integer;
begin
   d:=Detect;
   InitGraph(d,m,'C:\TP7\BGI');
   error:=GraphResult;
   if error<>0 then begin
```

```
      WriteLn(GraphErrorMsg(error));{Повідомлення про помилку}
      Halt
   end;
   WriteLn(GetDriverName);                    {Ім'я драйвера}
   WriteLn(GetModeName(GetGraphMode));{Повідомлення про режим}
   WriteLn(GetMaxMode);    {Максимально можливий номер режиму}
   GetModeRange(CurrentDriver,l,h);      {Діапазон режимів...}
   WriteLn(l:5,h:5);               {...для активного драйвера}
   ReadLn;
   {Робота в графічному режимі}
   CloseGraph;
   {Робота в текстовому режимі}
end.
```

Якщо графічний драйвер завантажений (а це має місце після ініціалізації графіки до виконання процедури **CloseGraph**), то можна переключати його режим за допомогою процедури

SetGraphMode(режим**)**

з одним параметром типу **Integer**, який задає номер графічного режиму. Цією ж процедурою можна скористатися для повернення в графічний режим після тимчасового виходу в текстовий режим. Власне тимчасове відновлення текстового режиму провадиться процедурою без параметрів

RestoreCRTMode.

При поверненні в графічний режим після виходу з нього по **RestoreCRTMode** можна змінити режим роботи графічного драйвера (але не сам драйвер).

Процедура **SetGraphMode** при переключенні системи в графічний режим викликає процедуру без параметрів

GraphDefaults,

яка задає графічні параметри за замовчуванням.

```
{Приклад 11.3}
{Здійснити тимчасовий вихід у текстовий режим і повернутися
 назад}
uses Graph;
var
  d,m,m1:Integer;
begin
    d:=Detect;
    InitGraph(d,m,'C:\tp7\bgi');
    OutText('Натисни <Enter> для RestoreCRTMode');ReadLn;
    m1:=GetGraphMode;
    RestoreCRTMode;
    WriteLn('Текстовий режим');
```

```
    WriteLn('Для повернення у графіку натисни <Enter>');
    ReadLn;
    SetGraphMode(m1);
    OutText('Натисни <Enter>');ReadLn;
    CloseGraph;
end.
```

Оскільки номер поточного графічного режиму процедура **InitGraph** записує у свій другий параметр, то в програмі оператор m1:=GetGraphMode фактично зайвий, бо замість m1 можна використовувати m.

11.2. Графічні вікна. Координати точки. Графічний курсор

Як і в текстовому режимі, у графіці робота з екраном здійснюється у вікні. При установці графічного режиму за замовчуванням графічним вікном буде весь екран. Верхній лівий кут графічного вікна визначається як (0,0). Значення *X* (номер стовпчика) збільшується вправо, значення *Y* (номер рядка) – униз. Графічна система підтримує поняття поточного покажчика (CP – Current Pointer), що є подібним курсору в текстовому режимі, але на екрані не висвітлюється. Процедура **InitGraph** переміщує CP у початок координат.

Функції

GetMax

і

GetMax

повертають максимальні горизонтальну і вертикальну координати графічного екрана.

Графічне вікно можна змінити за допомогою процедури

SetViewPort(X1**,** Y1**,** X2**,** Y2**,** ознака_відсічення**),**

що встановлює графічне вікно у вигляді прямокутника з координатами протилежних кутів (X1,Y1) і (X2,Y2) і сторонами, рівнобіжними осям координат екрана. Параметри X1, Y1, X2, Y2 мають тип **Integer**. П'ятий параметр (типу **Boolean**) визначає, чи повинні відсікатися частини малюнка, що виходять за межі вікна. Константи **ClipOff** (значення **False**) і **ClipOn** (значення **True**), задані замість останнього параметра, вказують відповідно на те, що відсічення не повинно провадитись і повинно провадитись. Графічний курсор переводиться в лівий верхній кут створеного вікна.

Після створення графічного вікна всі координати (за винятком **SetViewPort**) визначаються в цьому вікні.

Для роботи з графічним курсором у модулі **Graph** введені функції **GetX** і **GetY** і процедури **MeveTo** і **MoveRel**.

Функції без параметрів

GetX

і

GetY

повертають горизонтальну і вертикальну координати графічного курсора в системі координат поточного графічного вікна.

Процедура

MoveTo(X**,** Y**)**

переміщує графічний курсор у точку (X,Y), де X і Y – вирази типу **Integer**, а процедура

MoveRel(dx**,** dy**)**

з параметрами типу **Integer** забезпечує переміщення графічного покажчика на dx пікселів по осі X і dy пікселів по осі Y.

Графічне вікно в будь-який момент може бути очищене процедурою без параметрів

ClearViewPort,

яка заповнює вікно кольором першої позиції палітри і переводить графічний курсор у лівий верхній кут графічного вікна. Якщо виникає помилка, то результатом функції **GraphResult** буде GrNoScanMem=-6 – недостача пам'яті для заповнення області.

Подібну дію виконує й процедура без параметрів

ClearDevice,

яка очищує весь екран, заливаючи його кольором фона, і встановлює графічний курсор у лівому верхньому куті поточного графічного вікна.

Щоб довідатися про характеристики поточного вікна, використовують процедуру

GetViewSettings(характеристики_вікна**),**

яка у своєму єдиному **var**-параметрі повертає координати вікна й ознаку відсічення. Тип параметра цієї процедури визначений у модулі **Graph** у такий спосіб:

```
type
  ViewPortType=record
    X1,Y1,X2,Y2:Integer; {Координати кутів вікна}
```

```
    Clip:Boolean                    {Ознака відсічення}
  end;
```

Процедура **InitGraph** скидає всі графічні параметри (поточний покажчик, палітру, колір, ...) і встановлює для них значення за замовченням.

11.3. Побудова точки і відрізків прямих

Для побудови точки служить процедура

PutPixel(X**,** Y**,** колір**).**

Вона малює піксель з координатами (X,Y) кольором, номер якого заданий третім параметром (тип **Word**).

Для промальовування відрізків прямих найчастіше використовується процедура

Line(X1**,** Y1**,** X2**,** Y2**),**

що виводить на екран відрізок прямої лінії між точками (X1,Y1) і (X2,Y2), координати яких задаються значеннями типу **Integer**. Положення графічного покажчика при цьому не змінюється.

Процедура

LineRel(dx**,** dy**)**

з параметрами типу **Integer** здійснює малювання відрізка прямої лінії, який з'єднує точку, що зазначена графічним курсором, з точкою на відстані dx по горизонталі і dy по вертикалі, а процедура

LineTo(X, Y**)**

малює відрізок прямої, що з'єднує точку, зазначену графічним курсором, із точкою, яка має координати (X,Y).

При виконанні процедур **LineRel** і **LineTo** графічний курсор переміщується в кінцеву точку відрізка.

```
{Приклад 11.4}
{Побудувати один період синусоїди з амплітудою на весь екран.
 Вісь X повинна проходити горизонтально в середині екрана;
 вісь Y орієнтувати знизу вгору. Зобразити координатні
 осі з вказівкою стрілок на кінцях}
uses Crt,Graph;
var
   dr,m:Integer;
   maxX,maxY,x,y:Integer;
   s,t:Real;
```

```
begin
   dr:=Detect;
   InitGraph (dr,m,'C:\tp7\bgi');
   if GraphResult<>grOk then Halt;
   max:=GetMax;max:=GetMax;
   t:=0;              {Завдання початкового значення аргументу}
   for x:=10 to maxX-10 do                {Перебір значень ...}
   begin           {... координати X від X=10 до X=GetMaxX-10}
      s:=Sin(t)*((maxY-10) div 2);              {Масштабування}
      y:=max div 2-Trunc(s);          {Обчислення координати Y}
      t:=t+2*Pi/(maxX-20);                 {Зміна значення ...}
                  {...аргументу при зміні екранного X на 1}
      PutPixel(x,y,Blue);                   {Вивід синьої точки}
   end;
   Line(5,max div 2,maxX-10,max div 2);           {Гориз. вісь}
   Line(10,maxY-5,10,5);                           {Верт. вісь}
   MoveTo(10,5);            {Курсор у верхній кінець верт. осі}
   LineTo(13,10);              {Малюємо праву половину стрілки}
   MoveTo(10,5);  {Курсор перемістився; повертаємо його назад}
   LineTo(7,10);                {Малюємо ліву половину стрілки}
   MoveTo(maxX-10,max div 2);      {Курсор у кінець гориз. осі}
   LineRel(-5,-3);            {Малюємо верхню половину стрілки}
   MoveRel(5,3);  {Курсор перемістився; повертаємо його назад}
   LineRel(-5,3);              {Малюємо нижню половину стрілки}
   repeat until KeyPressed;
   CloseGraph;
end.
```

11.4. Вибір і запам'ятовування типу лінії

TP дозволяє використовувати лінії різних типів, у тому числі лінії, обумовлені користувачем. Для цього служить процедура

SetLineStyle(стиль**,** шаблон**,** товщина**),**

усі три параметри якої мають тип **Word**.

Перший параметр задає тип лінії. Його можливі значення визначають константи, описані в модулі **Graph**:

SolidLn=0	– суцільна лінія;
DottedLn=1	– пунктирна лінія (із крапок);
CenterLn=2	– осьова (штрихпунктирна) лінія;
DashedLn=3	– штрихова лінія;
UserBitLn=4	– лінія, обумовлена користувачем.

При значенні **UserBitLn** тип лінії вибирається за системою бітів, що задається в другому параметрі; у іншому випадку береться стандартна лінія для кожного зі значень, а другий параметр ігнорується. Використання шаблона для типу лінії забезпечує побудову лінії з фрагментів по 16 пікселів. Цей шаблон повторюється по всій довжині лінії. Якщо в якому-небудь біті шаблона записана одиниця, то це відповідає точці на фрагменті лінії, якщо 0, то пропуску точки.

Третій параметр задає товщину лінії. Для нього в модулі **Graph** введені дві константи:

NormWidth=1 – тонка лінія;

ThickWidth=3 – товста лінія.

```
{Приклад 11.5}
{Програма, яка ілюструє деякі типи ліній.
 Четверні точки виходять при значенні шаблона, що
 дорівнює 15 (1111 у двійковій системі числення)}
uses Graph;
var
   d,m:Integer;
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>grOk then begin
      Write('Помилка');Halt
   end;
   SetLineStyle(SolidLn,0,NormWidth);        {Нормальна лінія}
   Line(5,5,100,5);
   SetLineStyle(DottedLn,0,1);                 {Лінія з точок}
   Line(5,25,100,25);
   SetLineStyle(UserBitLn,15,1);     {Лінія з четверних точок}
   Line(5,45,100,45);
   SetLineStyle(CenterLn,0,3); {Штрих-пунктирна товста лінія}
   Line(5,65,100,65);
   ReadLn;
   CloseGraph;
  end.
```

Параметри лінії можуть бути запам'ятовані з метою подальшого використання. Для цього служить процедура

GetLineSettings(інформ_про_лінію**).**

Її єдиний параметр має тип **LineSettingsType**, що визначений у модулі **Graph** у такий спосіб:

```
type
    LineSettingsType=record
                          LineStyle:Word;
                          Pattern  :Word;
                          Thickness:Word;
    end;
```

У поле **LineStyle** записується номер вигляду лінії (від 0 до 4). Якщо він дорівнює **UserBitLn** (тобто 4), то в поле **Pattern** заноситься вигляд лінії (шаблон з 16 біт, заданий користувачем). Поле **Thickness** служить для запам'ятовування товщини лінії.

```
{Приклад 11.6}
{Запам'ятовування і відновлення параметрів лінії}
uses Graph;
var
   d,m:Integer;
   LineInfo:LineSettingsType;
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   SetLineStyle(UserBitLn,31,ThickWidth); {Установлюємо стиль}
   Line(0,0,GetMax,GetMax);
   GetLineSettings(LineInfo);          {Запам'ятовуємо стиль}
   SetLineStyle(SolidLn,0,NormWidth);        {Змінюємо стиль}
   Line(0,GetMax,GetMax,0);
   {Продовження програми}
   with LineInfo do                          {Відновлюємо стиль}
     SetLineStyle(LineStyle,Pattern,Thickness);
   Line(GetMaxX div 2,0,GetMaxX div 2,GetMaxY);
   ReadLn;
   CloseGraph
end.
```

Якщо не передбачено інше, то виведені лінії просто накладаються на існуюче зображення. За допомогою процедури

SetWriteMode(спосіб**)**

з параметром типу **Word** можна змінити спосіб виводу ліній. Якщо її параметр має значення 0, то вивід ліній відбувається звичайним чином; значення ж параметра рівне 1 указує на те, що надалі в точках перетинання лінії й зображення колір пікселів визначається значенням, яке дорівнює результату виконання бітової операції **xor** над кольором ліні, що виводиться, і кольорами існуючого зображення. У модулі **Graph** для процедури **SetWriteMode** введені

дві константи: **CopyPut**=0 і **XORPut**=1. Дворазовий вивід лінії в режимі **XORPut** не змінює зображення на екрані.

Дія процедур **SetLineStyle** і **SetWriteMode** поширюється на процедури **Line**, **LineRel**, **LineTo**, **Rectangle** і **DrawPoly**.

11.5. Завдання кольору фону і графіки

Для зміни кольору фону служить процедура

SetBkColor(номер_кольору**).**

Колір фону замінюється таким кольором, який у поточній палітрі має позицію з номером, що дорівнює значенню параметра. На відміну від процедури **TextBackGround**, яка у текстовому режимі встановлює колір фону для подальшого виводу, не змінюючи поточного зображення, виконання процедури **SetBkColor** спричиняє зміну фону для поточного графічного зображення. Кольори задаються константами:

Black=0	– чорний,	**DarkGray**=8	– темно-сірий,
Blue=1	– синій,	**LightBlue**=9	– світло-синій,
Green=2	– зелений,	**LightGreen**=10	– світло-зелений,
Cyan=3	– блакитний,	**LightCyan**=11	– світло-блакитний,
Red=4	– червоний,	**LightRed**=12	– світло-червоний,
Magenta=5	– фіолетовий,	**LightMagenta**=13	– бузковий,
Brown=6	– коричневий,	**Yellow**=14	– жовтий,
LightGray=7	– світло-сірий,	**White**=15	– білий.

Відзначимо, що в графіці колір фону – це кожний з 16 кольорів, на відміну від текстового режиму, де фоновими можуть бути тільки перші 16 кольорів.

Вибір поточного кольору графіки здійснюється процедурою

SetColor(номер_кольору**).**

Функції без параметрів **GetBkColor**, **GetColor** і **GetMaxColor** повертають відповідно номер кольору в палітрі для кольору фону, номер кольору в палітрі для кольору графіки і максимальний номер кольору.

```
{Приклад 11.7}
{Побудувати два квадрати зі штрихових і штрихпунктирних
 ліній різного кольору. Здійснювати автоматичну зміну кольору
 фону до натискання будь-якої клавіші}
```

```
uses Graph,Crt;
var
   d,m:Integer;
begin
   d:=0;
   InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>grOk then  Write('Помилка')
   else begin                        {Колір фону за замовчуванням}
      SetColor(Green);                   {Колір графіки - зелений}
      MoveTo(10,10);
      SetLineStyle(3,0,2);                        {Штрихова лінія}
      LineRel(0,80); LineRel(80,0);
      LineRel(0,-80);LineRel(-80,0);
      SetColor(Magenta);                       {Фіолетова графіка}
      MoveRel(100,0);
      SetLineStyle(2,0,1);                 {Штрих-пунктирна лінія}
      LineRel(0,80);LineRel(80,0);
      LineRel(0,-80);LineRel(-80,0);
      repeat
        Delay(1000);
        SetBkColor(Yellow);                             {Жовтий фон}
        Delay(500);
        SetBkColor(LightGray);                  {Світло-сірий фон}
      until KeyPressed
   end
end.
```

За замовчуванням для фону береться колір з максимально можливим номером, а для графіки – з мінімальним.

Для аналізу кольору точок екрану використовується функція

GetPixel(X,Y**),**

що повертає у вигляді значення типу **Word** номер кольору пікселя з координатами (X,Y).

Для одержання інформації про колір у модулі **Graph** введені ще три функції без параметрів:

GetBkColor,
GetColor

і

GetMaxColor,

які у вигляді значень типу **Word** подають інформацію відповідно про номер кольору фону, про номер кольору графіки і про максимальний номер кольору, що може бути використаний у процедурі **SetColor**.

Приклад використання функцій **GetBkColor** і **GetColor** наведений у підрозділі 11.7 (приклад 11.10).

11.6. Зафарбування областей

Зафарбування областей здійснюється за шаблоном, за який використовується бітовий прямокутник 8×8 пікселів. Одиничний розряд у відповідному біті вказує на зафарбування точки в заданий колір. Для цих цілей створюється масив з 8 елементів типу **Byte**, що яким-небудь чином заповнюється нулями й одиницями.

Завдання шаблону для заповнення областей здійснюється процедурою

SetFillPattern (шаблон**,** номер_кольору**)**

Перший параметр вказує шаблон. Він повинен бути описаний з типом **FillPatternType**, який у модулі **Graph** визначений так:

```
type
   FillPatternType=array[1..8] of Byte
```

Другий параметр — це номер кольору шаблона.

Для формування шаблону необхідно задати на полі 8×8 пікселів візерунок з нулів і одиниць, де одиниці відповідають підсвіченим точкам. Далі рядки цього візерунка інтерпретуються як двійкові числа, що після їх переведення в шістнадцяткову чи десяткову систему числення записуються як елементи масиву з типом **FillPatternType**. Цей масив і повинен бути використаний як шаблон у процедурі **SetFillPattern**. Так, масив з оголошенням

```
const
  pat:FillPatternType=(0,0,0,$20,$20,$F8,$20,$20)
```

визначає шаблон заливання «у дрібний хрестик».

Для заповнення деякої області заданим шаблоном використовується процедура

FloodFill(X**,** Y**,** колір_межі**)**

Ця процедура заповнює шаблоном область, яка охоплює точку з координатами (X,Y) і обмежена лінією, номер кольору якої зазначений у третьому параметрі. Якщо лінія межі незамкнута, то буде залитий весь екран.

```
{Приклад 11.8}
{Зобразити квадрат, заповнений ромбами}
uses Graph;                                        {Шаблон: 00000000}
var                                                        {00010000}
   d,m,i,dx:Integer;                                       {00101000}
const                                                      {01000100}
   Pattern:FillPatternType=                                {10000010}
    ($0,$10,$28,$44,$82,$44,$28,$10);                      {01000100}
begin                                                      {00101000}
   d:=Detect;                                              {00010000}
   InitGraph(d,m,'c:\tp7\bgi');
   dx:=100;
   MoveTo((GetMaxX-dx)div 2,(GetMaxY-dx) div 2);
   for i:=1 to 2 do begin
      LineRel(dx,0);LineRel(0,dx);
      dx:=-dx;
   end;                                          {Установка шаблону...}
   SetFillPattern(Pattern,Red);          {... і кольору заливання}
   FloodFill(GetMaxX div 2+1,GetMaxY div 2+1,GetColor);
   ReadLn;
   CloseGraph
end.
```

При заливанні може використовуватися стандартний шаблон. Для його установки служить процедура

SetFillStyle(номер_шаблона**,** номер_кольору**).**

Тут перший параметр задає номер стандартного шаблона. Стандартні шаблони визначаються константами модуля **Graph**:

EmptyFill=0 – заповнення кольором фону;
SolidFill=1 – суцільне заповнення;
LineFill=2 – заповнення товстими горизонтальними лініями;
LtSlashFill =3 – заповнення похилими лініями(///);
SlashFill=4 – заповнення товстими похилими лініями(///);
BkSlashFill=5 – заповнення товстими косими лініями(\\\);
LtBkSlashFill=6 – заповнення косими лініями(\\\);
HatchFill=7 – заповнення вертикальною сіткою;
XHatchFill=8 – заповнення похилою сіткою;
InterLeaveFill=9 – заповнення частою косою сіткою;
WideDotFill=10 – заповнення точками;
CloseDotFill =11– заповнення точками (щільне);

Наприклад, якщо необхідно зобразити квадрат, заповнений похилою сіткою, то можна в наведеній вище програмі замінити

оператор **SetFillPattern** оператором SetFillStyle(XHatchFill,Red), не проектуючи власний шаблон у змінній Pattern.

Для запам'ятовування поточного шаблону, спроектованого користувачем, служить процедура

GetFillPattern(шаблон**).**

Її єдиний параметр — це **var**-параметр типу **FillPatternType**, у який заноситься поточний користувальницький шаблон. За замовчуванням окремі елементи шаблону мають значення $ff (всі одиниці) – повне зафарбування.

Подібну з процедурою **GetFillPattern** дію виконує процедура **GetFillSettings**. Однак якщо процедура **GetFillPattern** запам'ятовує шаблон, створений користувачем, то процедура **GetFillSettings** запам'ятовує колір і номер стандартного шаблона. Вона має один **var**-параметр і викликається так:

GetFillSettings(інформація_про_заливання**).**

Тип параметра цієї процедури визначений так:

```
type
   FillSettingsType=record
                          Pattern:Word;
                          Color  :Word;
   end;
```

Поле **Pattern** вказує номер шаблона, встановленого за допомогою процедури **SetFillStyle** (значення від 0 до 11) або процедури **SetFillPattern** (значення 12); поле **Color** вказує номер кольору.

Взаємодія процедур **GetFillSettings**, **GetFillPattern**, **SetFillStyle** і **SetFillPattern** ілюструє наведений нижче фрагмент програми.

```
{Приклад 11.9}
{Взаємодія процедур зчитування й установки шаблонів}
uses Graph;
var
   d,m:Integer;
   FillInfo:FillSettingsType;
   pat:FillPatternType;
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   {...}
   GetFillSettings(FillInfo); {Запам'ятовуємо стиль заливання}
   if FillInfo.Pattern=12 then    {Якщо шаблон користувача...}
     GetFillPattern(pat);          {...то запам'ятовуємо його}
   {...}
```

```
   with FillInfo do
     if Pattern=12 then      {Якщо був шаблон користувача,...}
         SetFillPattern(pat,Color)       {...відновлюємо його}
     else SetFillStyle(Pattern,Color);{Інакше був запам'я-...}
   {...}     {...тований стандартний шаблон; відновлюємо його}
end.
```

11.7. Рисування прямокутників, паралелепіпедів і багатокутників

Для рисування прямокутників зі сторонами, рівнобіжними межам екрана, у модулі **Graph** введені дві процедури:

Bar(X1**,**Y1**,**X2**,**Y2**)**

і

Rectangle(X1**,**Y1**,**X2**,**Y2**),**

які відповідно будують зафарбований і незафарбований прямокутник з лівим верхнім кутом у точці (X1,Y1) і правим нижнім – у точці (X2,Y2). Процедура **Bar** веде зафарбовування шаблоном і кольором, встановленими за допомогою **SetFillPattern** чи **SetFillStyle**, не обводячи прямокутник, а процедура **Rectangle** рисує прямокутник поточним кольором.

```
{Приклад 11.10}
{Зобразити зафарбований прямокутник, що переміщується}
uses Graph,Crt;
var
   d,m,Color,x,y,dx:Integer;       {dx – приріст координати x}
const
   Pattern:FillPatternType=(0,0,0,$20,$20,$F8,$20,$20);
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>0 then WriteLn('Помилка')
   else begin
      x:=0;y:=GetMax div 2-50;dx:=1;
      SetBkColor(Blue);                              {Колір фону}
      repeat
        Color:=Yellow;                          {Колір заливання}
        SetFillPattern(Pattern,Color);   {Параметри заливання}
        SetColor(Color);   {Кольори межі і заливання однакові}
        Bar(x,y,x+100,y+100);{Рисуємо зафарбований прямокутн.}
        SetColor(GetBkColor);    {Тепер рисуємо кольором фону}
        SetFillStyle(EmptyFill,GetColor);     {Стандартний...}
        Bar(x,y,x+100,y+100);     {...шаблон: заливання фоном}
```

```
      x:=x+dx;          {Зміна координати, а нижче напрямку}
      if x=0 then dx:=1 else if x=GetMaxX-100 then dx:=-1;
    until KeyPressed;
    CloseGraph;
  end;
end.
```

Процедура

Bar3d(X1**,** Y1**,** X2**,** Y2**,** глибина**,** верхня_грань**)**

рисує паралелепіпед, передня грань якого зафарбовується за шаблоном. Ліва верхня і права нижня вершини передньої грані визначаються точками (X1,Y1) і (X2,Y2). Глибина паралелепіпеда в пікселях задається п'ятим параметром, який має тип **Integer**. Останній параметр (тип **Boolean**) вказує на те, треба (значення **True**) чи не треба (значення **False**) рисувати верхню грань. Для завдання цього параметра в модулі **Graph** введені дві константи: **TopOn** (значення **True)** і **TopOff** (значення **False)**. Якщо рисунок не вийде, буде помилка **grNoScanMem**=-6.

```
{Приклад 11.11}
{Побудувати паралелепіпед з білою гранню, а поруч два
 розташованих один на одному паралелепіпеди з зеленою
 і червоною гранями}
uses Crt,Graph;
const Pattern:FllPatterntype=
       ($38,$6C,$C6,$92,$C6,$6C,$38,$0);
var
   d,m,cv:Integer;
begin
   d:=Detect;
   InitGraph(d,m,'C:\tp7\bgi');
   if GraphResult<>grOk then Write('Помилка')
   else begin
      SetLineStyle(SolidLn,0,ThickWidth);
      SetColor(Blue);
      Bar3d(180,200,280,300,20,TopOn);        {Заливання за...}
                                             {...замовчуванням}
      SetLineStyle(DashedLn,0,ThickWidth);
      SetFillPattern(Pattern,Green);{Шаблон і колір заливання}
      Bar3d(330,100,430,200,30,TopOn);{Верхня грань рисується}
      SetFillPattern(Pattern,Red);  {Шаблон і колір заливання}
      Bar3d(330,200,430,300,30,TopOff);       {Верхня грань...}
   end;                                       {...не рисується}
   repeat until KeyPressed;
   CloseGraph
end.
```

Процедура

DrawPoly(кількість_точок**,** координати_точок**)**

служить для побудови ламаної, яка з'єднує задану кількість точок, координати яких (X,Y) у вигляді пар цілих чисел ідуть одна за одною у частині пам'яті, що виділена під змінну, котра зазначена як другий параметр. За цю змінну звичайно служить покажчик, але може використовуватися і масив цілого типу, в якому послідовно записуються координати вершин. Ця процедура не з'єднує першу й останню вершини, тому для побудови багатокутника (замкнутої ламаної) потрібно задати однакові координати для першої й останньої вершин.

```
{Приклад 11.12}
{Побудувати червоний шестикутник і жовту ламану,
 що з'єднує 7 вершин з випадковими координатами}
uses Crt,Graph;
const v:array[1..7,1..2] of Integer= ((100,100),(160,120),
         (160,180),(100,200),(40,180),(40,120),(100,100));
type
   t_coord=array[1..100] of PointType;
var
   d,m,i:Integer;
   coord:^t_coord;
begin
   d:=Detect;
   InitGraph(d,m,'C:\tp7\bgi');
   SetColor(Red);
   DrawPoly(7,v); {Використовується статичний масив координат}
   GetMem(coord,7*2*SizeOf(Integer));   {Відводимо пам'ять...}
   Randomize;                        {... під координати 7 точок}
   for i:=1 to 7 do
      with coord^[i] do begin             {Формуємо координати}
         x:=Random(GetMaxX+1);
         y:=Random(GetMaxY+1);
      end;
   SetColor(Yellow);
   DrawPoly(7,coord^);   {Використовуємо динамічний масив ...}
                                                 {... координат}
   repeat until KeyPressed;
   FreeMem(coord,7*2*SizeOf(Integer));
   CloseGraph
end.
```

У програмі в масиві v записані координати 7 вершин для того, щоб одержати замкнуту ламану. При оголошенні типу t_coord

використаний тип **PointType**, визначений у модулі **Graph** у такий спосіб:

```
type
   PointType=record
                X,Y:Integer
   end;
```

Подібна з **DrawPoly** процедура

FillPoly(кількість_вершин**,** координати_вершин**)**

будує і заповнює заданим шаблоном багатокутник із зазначеною кількістю вершин, координати яких знаходяться в області пам'яті, що виділена під другий параметр (див. вище).

```
{Приклад 11.13}
{Побудувати зафарбований шестикутник}
uses Graph;
const v:array[1..6,1..2] of Integer=
  ((100,100),(160,120),(160,180),(100,200),(40,180),(40,120));
var
  d,m:Integer;
begin
  d:=Detect;
  InitGraph(d,m,'C:\tp7\bgi');
  SetColor(Red);
  FillPoly(6,v);         {Заповнюємо шаблоном за замовчуванням}
  ReadLn;
  CloseGraph
end.
```

Тут шестикутник зафарбовується шаблоном за замовчуванням ($ff).

На відміну від **DrawPoly**, процедура **FillPoly** будує не ламану лінію, а багатокутник. Тому при звертанні до неї вказують кількість вершин багатокутника, а не кількість точок, які з'єднує ламана. При побудові багатокутника за допомогою **DrawPoly** перший параметр повинен бути на 1 більше значення першого параметра, що задається в процедурі **FillPoly** при побудові аналогічного багатокутника (порівняй дві програми).

11.8. Побудова кругів, кіл, еліпсів, дуг, секторів

Процедура

Circle(X**,** Y**,** радіус**)**

рисує коло заданого радіуса (тип **Word**) з центром у точці (X,Y). Радіус визначає кількість пікселів по горизонталі.

Рисування дуги кола з центром у точці (X,Y) і заданим радіусом здійснює процедура

Arc(X**,** Y**,** кут_початку**,** кут_кінця**,** радіус**),**

третій і четвертий параметри якої задають відповідно кут початку (у градусах) і кут кінця (у градусах) дуги, причому значення кута 0 відповідає напрямку годинної стрілки на 3 год. Рисування відбувається проти годинникової стрілки.

За допомогою процедури

GetArcCoords(координати_дуги**)**

можна визначити координати останньої дуги, побудованої за допомогою процедури **Arc**. Її параметр має тип **ArcCoordsType**, що описаний у модулі **Graph** так:

```
type
  ArcCoordsType=record
    X,Y:Integer;                    {Координати центру}
    Xstart,Ystart:Integer;{Координати початку дуги}
    Xend,Yend:Integer          {Координати кінця дуги}
  end;
```

Процедура

PieSlice(X**,** Y**,** кут_початку**,** кут_кінця**,** радіус**)**

зображує сектор круга заданого радіуса. Параметри процедури збігаються з параметрами процедури **Arc**. Сектор заповнюється поточним шаблоном і обводиться з усіх боків лінією при установці типу лінії **SolidLn**; при інших типах лінії прорисовуються тільки радіуси без креслення дуги. Якщо третій і четвертий параметри дорівнюють 0º і 360º, зображується круг. Найчастіше ця процедура застосовується при побудові секторних діаграм. Дія процедури **GetArcCoords** поширюється і на **PieSlice**.

```
{Приклад 11.14}
{Побудувати червону окружність і жовту дугу окружності з
 початковим кутом 0 і кінцевим кутом 90 градусів. Продовжити
 дотичними кінці дуги. Зобразити сектор, залитий зеленими
 горизонтальними лініями}
```

```
uses Graph;
var
   d,m:Integer;
   coords:ArcCoordsType;
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>0 then WriteLn('Помилка')
   else begin
      SetBkColor(Blue);
      SetColor(Red);
      Circle(100,100,50);            {Товщина за замовчуванням}
      SetColor(Yellow);
      SetLineStyle(CenterLn,0,DashedLn);            {Перший...}
      Arc(200,200,0,90,50);       {...параметр не враховується}
      GetArcCoords(coords);
      with coords do begin                  {А тут враховується}
         Line(Xstart,Ystart,Xstart,Ystart+50);
         Line(Xend,Yend,Xend-50,Yend)
      end;
      SetFillStyle(LineFill,Green);
      PieSlice(300,200,0,90,50);                {Сектор без...}
      ReadLn;                             {...прорисовування дуги}
      CloseGraph
   end
end.
```

Процедура

Ellipse(X**,** Y**,** кут_початку**,** кут_кінця**,** X_радіус**,** Y_радіус**)**

будує дугу еліпса з центром у точці (X,Y) для заданих третім і четвертим параметрами початкового і кінцевого кутів з довжинами півосей по X і Y, визначеними останніми двома параметрами. Перші два параметри мають тип **Integer**, інші – **Word**. Вимір і напрямок відліку кутів виробляється так само, як і в процедурі **Arc**.

Процедури **Circle**, **Arc** і **Ellipse** вичерчують криві поточним кольором тільки суцільною лінією, товщина якої визначається значенням третього параметра процедури **SetLineStyle**.

Процедура

Sector(X**,** Y**,** кут_початку**,** кут_кінця**,** X_радіус**,** Y_радіус**)**

з параметрами, аналогічними параметрам процедури **Ellipse**, зображує сектор еліпса, заповнений поточним шаблоном. Облямівка сектора здійснюється аналогічно тому, як це робиться в **PieSlice**. Якщо третій

параметр дорівнює 0, а четвертий 360, то зображується заповнений еліпс.

Щоб одержати заповнений еліпс, обведений лінією, використовується процедура

FillEllipse(X**,** Y**,** X_радіус**,** Y_радіус**).**

Її перший і другий параметри — це координати центра (тип **Integer**), а третій і четвертий — довжини півосей по осях X і Y відповідно (у пікселях, тип **Word**). Заливання виробляється поточним шаблоном, а лінія зображується відповідно до установок процедур **SetColor** і **SetLineStyle** (причому перший параметр **SetLineStyle** ігнорується і покладається рівним **SolidLn** – суцільна лінія).

```
{Приклад 11.15}
{Побудувати еліпс і дугу еліпса від 50 до 270 градусів}
{Побудувати заповнений сектор еліпса і заповнений еліпс}
uses Graph;
var
   d,m:Integer;
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>0 then WriteLn('Помилка')
   else begin
      Ellipse(100,100,0,360,50,75);           {Еліпс}
      Ellipse(250,100,50,270,75,30);   {Дуга еліпса}
      SetColor(Yellow);
      Sector(450,100,50,270,75,30);  {Сектор еліпса}
      FillEllipse(450,250,50,70); {Заповнений еліпс}
      ReadLn;
      CloseGraph
   end
end.
```

11.9. Визначення коефіцієнта стиску екрана

Адаптери VGA і SVGA характерні тим, що в них відстані між сусідніми пікселями по вертикалі і горизонталі збігаються. В інших адаптерах це не так. За замовчуванням при побудові кіл рисування відбувається без перекручувань, оскільки в цьому випадку фактично задається радіус за X, а координата Y кожного пікселя зображення перераховується з урахуванням коефіцієнта стиску (перекручування). Спроба побудови квадрата без врахування коефіцієнта перекру-

чування призводить до того, що насправді буде зображений прямокутник. Щоб одержати коефіцієнт перекручування, потрібно скористатися процедурою

GetAspectRatio(вигляд_за_X**,** вигляд_за_Y**),**

що повертає два значення типу **Word**, відношення яких відповідає поточному коефіцієнту стиску (за замовчуванням друге значення дорівнює 10000). Ці два значення можуть бути використані при побудові зображень.

```
{Приклад 11.16}
{Для драйвера EGA побудувати два квадрати: один зі стороною
 50 пікселів за X, а другий - 50 пікселів за Y}
uses Graph;
var
  d,m,x,y:Integer;
  xAsp,yAsp:Word;
begin
  d:=EGA;m:=EGAHi;
  InitGraph(d,m,'C:\tp7\bgi');
  GetAspectRatio(xAsp,yAsp); {Зчитуємо параметри коеф. стиску}
  x:=30;y:=GetMax div 2;
  Rectangle(x,y,x+50,y+Round(50.0*xAsp/yAsp));    {1-й квадрат}
  x:=x+100;
  Rectangle(x,y,x+Round(50.0*yAsp/xAsp),y+50);    {2-й квадрат}
  ReadLn;
  CloseGraph
end.
```

За допомогою процедури

SetAspectRatio(вигляд_за_X**,** вигляд_за_Y**)**

можна установити новий коефіцієнт стиску.

Зміною коефіцієнта стиску можна забезпечити побудову зображень, перекручених стосовно зображення, що утворюється при стандартному коефіцієнті стиску.

```
{Приклад 11.17}
{Установлення коефіцієнта стиску. Рисується коло.
 Стрілками <вліво>-<вправо>, <униз>-<вгору> зменшується
 та збільшується параметр коефіцієнта стиску за X і Y.
 відповідно. Остаточна установка - за <Enter>}
uses Crt,Graph;
var
  d,m:Integer;
  xAsp,yAsp,dx,dy:Word;
  ch:Char;
begin
  d:=Detect;
```

```
  InitGraph(d,m,'C:\tp7\bgi');
  GetAspectRatio(xAsp,yAsp);      {Зчитування параметрів...}
  dx:=0;dy:=0;      {...коефіцієнта стиску за замовчуванням}
  repeat
     SetColor(White);           {Рисуємо коло кольором фона}
     Circle(GetMaxX div 2,GetMaxY div 2,50);
     ch:=ReadKey;if ch=#0 then ch:=ReadKey;
     case ch of
       #72:Inc(dy);
       #75:if dx>0 then Dec(dx);
       #77:Inc(dx);
       #80:if dy>0 then Dec(dy);
     end;       {Нижче встановлюється новий коефіцієнт стиску}
     SetAspectRatio(xAsp+1000*dx,yAsp+1000*dy);
     SetColor(Red);
     Circle(GetMaxX div 2,GetMaxY div 2,50);
     repeat until KeyPressed;
  until ch=#13;
  repeat until KeyPressed;
  CloseGraph;
end.
```

11.10. Вивід тексту

Для виводу тексту в графічному режимі використовуються процедури **OutText** і **OutTextXY**. Процедура

OutText(рядок**)**

виводить біля графічного курсора рядок тексту, заданого у рядку-параметрі процедури. Шрифти визначаються за допомогою процедури **SetTextStyle**, колір – процедурою **SetColor**. Розміщення тексту щодо графічного курсора може бути визначено процедурою **SetTextJustify**. Якщо перший її параметр **LeftText**, а текст виводиться горизонтально, то горизонтальна координата курсора зміниться на ширину тексту. В інших випадках положення графічного курсора не зміниться. Якщо текст не поміщається в графічному вікні, він буде обрізаний на краях.

Процедура

OutTextXY(X**,** Y**,** рядок**)**

виводить біля точки з координатами (X,Y) рядок, який заданий у третьому параметрі. Положення графічного курсора не змінюється.

Числові дані попередньо повинні приводитися до рядкового подання за допомогою процедури **Str**.

Процедура

SetTextJustify(гориз**,** верт**)**

визначає спосіб вирівнювання виведеного тексту по горизонталі і по вертикалі. Вирівнювання здійснюється або щодо графічного курсора (при **OutText**), або щодо заданої точки (при **OutTextXY**). Горизонтальне вирівнювання (перший параметр) задається константами модуля **Graph**:

LeftText=0 – вирівнювання по лівому краю;
CenterText=1 – по центру;
RightText =2 – по правому краю.

Вирівнювання по вертикалі (другий параметр) здійснюється за допомогою констант:

BottomText=0 – по нижньому краю;
CenterText=1 – по центру;
TopText =2 – по верхньому краю.

Процедура

SetTextStyle(шрифт, напрямок, розмір)

з трьома параметрами типу **Word** задає тип шрифта (перший параметр), напрямок розташування тексту (другий параметр) і розмір символів (третій параметр).

Шрифт може бути заданий константами:

DefaulFont=0 – шрифт за замовчуванням, обумовлений шаблоном 8×8 біт (растровий шрифт);
TriplexFont =1 – потрійний штриховий шрифт;
SmallFont=2 – індексний штриховий шрифт;
SansSerifFont=3 – гротесковий штриховий шрифт;
GothicFont=4 – готичний штриховий шрифт.

Штрихові шрифти при збільшенні розміру букв не погіршуються. Вони зберігаються у файлах з розширенням .chr у тому ж каталозі, у якому зберігається bgi-драйвер. Шрифтам з номерами 1..4 відповідають файли trip.chr, litt.chr, sans.chr і goth.chr. Якщо відповідний файл відсутній, то підключається шрифт за замовчуванням. Українські (російські) літери стандартні штрихові шрифти не містять. У той же час, якщо комп'ютер може виводити українські (російські) літери, то шрифт за замовчуванням також буде містити їх. У TP 7.0 введені додаткові стандартні штрихові шрифти з номерами від 5 до 10, які містяться відповідно у файлах scri.chr, simp.chr, tsr.chr, lcom.chr, euro.chr, bold.chr. Можна також інсталювати нестандартний шрифт, для чого потрібно звернутися до функції

InstallUserFont(ім'я_файла**),**

параметром якої є ім’я файла, що містить шрифт (тип **string**). Функція у вигляді значення типу **Word** повертає привласнений шрифту номер. Більш 10 штрихових шрифтів TP не підтримує.

Напрямок виводу тексту задається константами:

HorizDir=0 – горизонтально зліва направо;

VertDir =1 – вертикально знизу нагору.

При вертикальному виводі текст читається знизу вгору, причому символи повернені на 90º проти годинникової стрілки. За замовчуванням вивід здійснюється горизонтально.

Значення 2 для другого параметра не регламентовано, але при використанні растрового шрифту вивід провадиться горизонтально зліва направо при повернених на 90º проти годинникової стрілки символах.

Розмір символів установлюється третім параметром. Для растрового шрифта діапазон розмірів задається значеннями від 1 до 10, для штрихових – від 4 до 32. Розмір, більший максимального, відповідає максимально можливому. При завданні розміру 0 установлюється мінімально можливий розмір.

За замовчуванням при виводі тексту по горизонталі позиція, щодо якої виводиться текст, фіксується в лівому верхньому куті тексту; при виводі по вертикалі – у правому верхньому куті.

Наведена нижче програма ілюструє різні напрямки виводу і способи вирівнювання.

```
{Приклад 11.18}
{Вивід тексту з різними способами вирівнювання
 і при різних напрямках виводу з використанням
 растрового шрифта. Щораз відзначати точку, біля
 якої провадився вивід}
uses Graph;
var
  d,m:Integer;
  h,v,p:Integer;
  sh,sv,sp:string;
begin
  d:=Detect;
  InitGraph(d,m,'C:\tp7\bgi');
  for h:=0 to 2 do      {Варіанти вирівнювання по горизонталі}
    for v:=0 to 2 do      {Варіанти вирівнювання по вертикалі}
      for p:=0 to 2 do begin        {Перебір напрямків виводу}
        SetColor(White);
        MoveTo(300,300);                {Установлюємо курсор}
        SetTextJustify(h,v); {Задаємо спосіб вирівнювання,...}
```

```
        SetTextStyle(0,p,1);       {...шрифт, напрямок, розмір}
        Str(h,sh);Str(v,sv);Str(p,sp);       {Підготовка даних}
        OutText('Hor='+sh+' Ver='+sv+' Direct='+sp);   {Вивід}
        SetColor(Yellow);
        Line(290,300,310,300);{Точка,біля котрої здійснювався}
        Line(300,290,300,310);  {вивід, стоїть на перетинанні}
        SetColor(Green);
        SetTextStyle(DefaultFont,HorizDir,1);
        SetTextJustify(LeftText,BottomText);
        OutTextXY(GetMax div 2-80,GetMax,'Press <Enter>');
        ReadLn;       {Вище був вивід у зазначених координатах}
        ClearDevice;
      end;
end.
```

Розтягання штрихових шрифтів по вертикалі і горизонталі здійснює процедура

SetUserCharSize(X_чисельн**,**X_знамен**,**Y_чисельн**,**Y_знамен**).**

Байтові параметри процедури задають коефіцієнти розтягання по горизонталі (X_чисельн/X_знамен) і вертикалі (Y_чисельн/Y_знамен). Процедура **SetTextStyle** при її виконанні скасовує дію процедури **SetUserCharSize**.

Для перевірки можливості виводу тексту потрібно знати кількість пікселів по вертикалі і горизонталі, що необхідна для виводу тексту. Це забезпечують дві функції:

TextHeight(рядок**)**

і

TextWidth(рядок**),**

які повертають відповідно вертикальний і горизонтальний розміри рядка, вказаного за парметр (у пікселях: тип **Word**).

Програма, що наводиться нижче, демонструє можливість використання описаних вище процедур і функцій для виводу текстових повідомлень у графічному режимі.

```
{Приклад 11.19}
{За допомогою клавіш зі стрілками вибирається розмір символів
 по вертикалі і горизонталі в шрифті TriplexFont. Остаточний
 вибір фіксувати по <Enter>. По <Esc> установлюється
 шрифт за замовчуванням. Наприкінці - контрольний вивід}
uses Graph,Crt;
var
   d,m:Integer;
   width,height:Word;
   xMult,xDiv,yMult,yDiv:Byte;
   ch:Char;
```

```
const
   result:string='Example';
   prom:string='Press <Left>,<Right>,<Up>,<Down> and <Enter>';
procedure user_size(var numerator,denominator:Byte);
begin                                              {Змінюється...}
   if numerator<255 then Inc(numerator) else  {...чисельник чи}
     if denominator>1 then Dec(denominator);     {...знаменник}
end;
begin
   d:=Detect;InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>0 then WriteLn('Помилка')
   else begin
      xMult:=32;xDiv:=32;yMult:=32;yDiv:=32;
      SetTextJustify(LeftText,BottomText);
      repeat
         ClearDevice;
         SetTextStyle(DefaultFont,HorizDir,1);
         width:=TextWidth(prom);   {Ширина підказки в пікселях}
                              {Вивід з урахуванням довжини рядка}
         OutTextXY((GetMaxX-width) div 2,GetMax,prom);
         SetTextStyle(1,HorizDir,0); {Устан. шрифт TriplexFont}
         SetUserCharSize(xMult,xDiv,yMult,yDiv);{Зміна розміру}
         width:=TextWidth(result);          {Ширина в пікселях}
         height:=TextHeight(result);        {Висота в пікселях}
      {Переміщуємо курсор з урахуванням ширини і висоти тексту}
         MoveTo((GetMaxX-width)div 2,(GetMax+height)div 2);
         OutText(result);                  {Вивід зразка тексту}
         ch:=ReadKey;if ch=#0 then ch:=ReadKey;
         case ch of
          #72:user_size(yMult,yDiv);  {Збільшення розміру по Y}
          #75:user_size(xDiv,xMult);   {Зменшення розміру по X}
          #77:user_size(xMult,xDiv);  {Збільшення розміру по X}
          #80:user_size(yDiv,yMult);   {Зменшення розміру по Y}
          #27:SetTextStyle(DefaultFont,HorizDir,1);{За замовч.}
         end;
      until (ch=#13)or(ch=#27);
      ClearDevice;
      width:=TextWidth(result);             {Ширина в пікселях}
      height:=TextHeight(result);           {Висота в пікселях}
      OutTextXY((GetMaxX-width)div 2,
                (GetMaxY+height)div 2,result);
      repeat until KeyPressed; CloseGraph
   end
end.
```

Параметри виводу тексту можуть бути запам'ятовані за допомогою процедури

GetTextSettings(параметри_виводу**)**

з **var**-параметром, тип якого описаний у модулі **Graph** як запис:

```
type
   TextSettingsType=record
                        Font:Word;
                        Direction:Word;
                        CharSize:Word;
                        Horiz:Word;
                        Vert:Word;
   end;
```

У полі **Font** запам'ятовується номер шрифта; поле **Direction** визначає розташування тексту (0 – горизонтально, 1 – вертикально); поле **CharSize** визначає масштаб символів (якщо його значення дорівнює 0 (константа **UserCharSize**), то вертикальні і горизонтальні розміри визначалися процедурою **SetUserCharSize**); поля **Horiz** і **Vert** визначають спосіб горизонтального і вертикального вирівнювання.

11.11. Робота зі сторінками

При роботі з моніторами EGA, VGA і Hercules можливе використання декількох сторінок з нумерацією від нуля. За замовчуванням активною є нульова сторінка (вона ж відображається на екрані). За допомогою процедури

SetActivePage(сторінка**)**

можна робити активною ту чи іншу сторінку, у результаті чого всі подальші операції будуть виконуватися на вказаній сторінці. За допомогою ж процедури

SetVisualPage(сторінка**)**

різні сторінки можна виводити на екран. В обох процедурах параметр — це номер сторінки у вигляді значення типу **Word**.

Отже, за допомогою процедури **SetActivePage** можна формувати зображення на кожній (навіть на невидимій) сторінці, а за допомогою процедури **SetVisualPage** забезпечується переключення сторінок на екран. Наприклад, послідовність операторів

```
SetActivePage(1);Line(100,100,200,200);SetVisualPage(1);
```

виконує такі дії: стає активною сторінка 1, при цьому видимою залишається сторінка 0; на невидимій сторінці 1 рисується лінія; здійснюється переключення екрана на сторінку 1; далі залишається активною і видимою сторінка 1.

На моніторах EGA і VGA багатосторінкова робота неможлива в режимі з максимальним номером. Тому, якщо ініціалізація графіки відбувалася в режимі автодетектування, необхідно змінити номер режиму. Наприклад, це може бути зроблено так:

```
if GetMode=GetMaxMode then SetGraphMode(GetMode-1);
```

11.12. Спрайти

TP дозволяє зчитувати з екрана (графічного вікна) зображення, що розміщується в деякій прямокутній області (спрайт), запам'ятовувати це зображення і переносити його в іншу частину екрана (вікна) чи переписувати у файл. При цьому зображення сприймається як шаблон, у якому деякі пікселі мають підсвічування, а деякі ні.

Процедура

GetImage(X1**,** Y1**,** X2**,** Y2**,** область_пам'яті**)**

у рамках графічного вікна виділяє прямокутну область з координатами протилежних вершин (X1,Y1) та (X2,Y2) і заносить шаблон, що міститься в ній, в область пам'яті, відведену під змінну, котра вказана як останній параметр (це може бути змінна будь-якого типу; важливо тільки, щоб її розмір дозволяв зберігати спрайт).

Процедура ж

PutImage(X**,** Y**,** область_пам'яті**,** метод_виводу**)**

розміщує у вікні прямокутне графічне зображення, задане в області пам'яті, виділеній під змінну, що вказана як третій параметр. При цьому верхній лівий кут прямокутника сполучається з точкою (X,Y). Накладення зображення виконується відповідно зі значенням останнього параметра (тип **Word**). Зображення на краях графічного вікна не обрізується. Виняток: якщо нижнім краєм вікна є нижній край екрана, то зображення, що виходить за цей край (і тільки за цей), обрізується.

Спосіб накладення зображення на екран обумовлюється відповідно зі значенням останнього параметра процедури:

NormalPut=0 – заміщення зображення спрайтом;

XORPut=1 – операція **xor** (над бітами спрайта і відповідної ділянки екрана виконується операція **xor**);

OrPut=2 – операція **or**;

AndPut=3 – операція **and**;

NotPut=4 – операція **not** (заміщення з виконанням операції **not** над спрайтом).

При режимі **XORPut** повторний вивід спрайта в тому самому місці приводить до відновлення початкового зображення.

Щоб встановити обсяг пам'яті (у байтах), необхідний для запам'ятовування графічного зображення в спрайті, можна скористатися функцією

ImageSize(X1**,**Y1**,**X2**,**Y2**),**

що повертає значення типу **Word**.

```
{Приклад 11.20}
{Якщо зроблено автодетектування відеоплати VGA чи
 EGA, сформувати на невидимій сторінці зображення і
 запам'ятати його фрагмент з координатами протилежних кутів
 (30,30), (79,79). Сформувати зображення на видимій
 сторінці. Вивести у верхній частині екрана запам'ятований
 спрайт і забезпечити його переміщення вниз до границі екрана}
uses Graph;
type
   t_bit_map=array[1..65535]of Byte;     {Тип для змінної,...}
var                          {...у якій буде зберігатися спрайт}
   d,m,y:Integer;
   is:Word;                                      {Розмір спрайта}
   bit_map,bit_map1:^t_bit_map;          {Змінні для спрайтів}
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   if (d=VGA)or(d=EGA) then begin {Можлива багатостор. робота}
      SetGraphMode(m-1);{Забезпечуємо багатосторінковий режим}
      SetActivePage(1);    {Активна (але невидима) сторінка 1}
                {Формуємо зображення на сторінці 1 (невидиме)}
         {...}
      is:=ImageSize(30,30,79,79);    {Зчитуємо розмір спрайта}
      GetMem(bit_map,is);       {Відводимо пам'ять під спрайт}
      GetImage(30,30,79,79,bit_map^);         {Зчитуємо спрайт}
      SetActivePage(0);   {Тепер активна(і видима) сторінка 0}
                   {Формуємо зображення на сторінці (видиме)}
         {...}
      y:=0;
      GetMem(bit_map1,is);      {Відводимо пам'ять під спрайт}
      repeat{Нижче запам'ятовуємо зображ. в області виводу...}
          GetImage(200,y,249,y+49,bit_map1^);     {...спрайта}
          PutImage(200,y,bit_map^,NormalPut);  {Вивід спрайта}
          PutImage(200,y,bit_map1^,NormalPut);       {Старе...}
          Inc(y);       {...зображення відновлене, змінюємо Y}
      until y>GetMax             {Поки не досягнута нижня межа}
```

```
   end
   else OutTextXY(100,200,'Не той драйвер');
   ReadLn;
   CloseGraph
end.
```

11.13. Установка палітри

Палітра — це максимальний набір кольорів, що підтримуються BGI-драйвером. Вона включає 16 кольорів для VGA і EGA як у текстовому, так і в графічному режимах.

Кольори нумеруються від 0 до 15. Ці значення називають програмними кольорами. Кожному з них відповідає апаратний колір з повної палітри.

Для адаптера EGA введені такі апаратні кольори:

EGABlack	= 0,	**EGADarkGray**	=56,
EGABlue	= 1,	**EGALightBlue**	=57,
EGAGreen	= 2,	**EGALightGreen**	=58,
EGACyan	= 3,	**EGALightCyan**	=59,
EGARed	= 4,	**EGALightRed**	=60,
EGAMagenta	= 5,	**EGALightMagenta**	=61,
EGABrown	=20,	**EGAYellow**	=62,
EGALightGray	= 7,	**EGAWhite**	=63.

Ці ж кольори використовуються й у VGA-палітрі.

Над кольорами палітри можна виконувати ряд операцій за допомогою засобів модуля **Graph**. Інформація про палітру міститься в деякій змінний, тип якої в модулі **Graph** задається так:

```
PaletteType=record
              Size:Byte;
              Colors:array[0..MaxColors] of ShortInt;
end;
```

де **Size** – число кольорів у палітрі (по суті, збільшене на 1 значення, що повертається функцією **GetMaxColor**); **Colors** – масив цілих чисел, які відповідають апаратним кольорам; **MaxColors** – константа, що задає максимальний номер кольору (оголошена в модулі **Graph** зі значенням 15).

За допомогою функції без параметрів

GetPaletteSize

можна взнати кількість кольорів у поточній палітрі (результат має тип **Integer**).

Процедура

GetDefaultPalette(палітра**)**

записує інформацію про встановлювану за замовчуванням палітру в змінну, задану як параметр, а процедура

GetPalette(палітра**)**

– про поточну палітру. Параметр в обох випадках має тип **PaletteType**.

Процедура

SetPalette (програмний_колір**,** апаратний_колір**)**

дозволяє встановити (змінити) один колір палітри. Її перший параметр – номер змінюваного програмного кольору, а другий – номер апаратного кольору, який ставиться у відповідність зазначеному програмному кольору.

```
{Приклад 11.21}
{Замінити в палітрі колір з номером 1 (синій) на колір з
 максимально можливим номером}
uses Graph;
var
  d,m:Integer;
  def_pal:PaletteType;
begin
    d:=Detect;
    InitGraph(d,m,'C:\tp7\bgi');
    GetDefaultPalette(def_pal);
    SetPalette(Blue,def_pal.Colors[GetMaxColor]);
    {Тепер синій колір відсутній, а при}
    {установленні кольору 1(Blue) буде}
    {установлюватися колір з максимальним}
    {номером, тобто Color(Blue) еквівалентний}
    {Color(GetMaxColor) };
    {...}
    CloseGraph;
end.
```

Якщо потрібно відновити палітру, запам'ятовану за допомогою процедур **GetPalette** чи **GetDefaultPalette**, або змінити кольори всієї палітри, то можна скористатися процедурою

SetAllPalette(палітра**).**

Як параметр найчастіше використовують змінну стандартного типу **PaletteType**, хоча можна використовувати змінну довільного типу, що має змінну довжину. У першому її байті повинна бути записана кількість кольорів N у встановлюваній палітрі, а наступні N байт – номери апаратних кольорів, які будуть використовуватися;

якщо задати колір з номером –1, то це означає, що відповідний програмний колір не міняється.

```
{Приклад 11.22}
{Палітрова мультиплікація: За допомогою зміни кольору в паліт-
 рі здійснюється обертання зафарбованого сектора, який змінює
 свій колір. Закінчення роботи - натискання будь-якої клавіші}
uses Graph,Crt;
var
   d,m,k:Integer;
   def_pal,pal:PaletteType;
   c:Byte;
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   GetDefaultPalette(def_pal);          {Зчитуємо початкову палітру}
   with pal do begin                   {Підготовлюємо дані для...}
      for  c:=Black to White do     {...заміни всіх кольорів...}
          Colors[c]:=EGAWhite;              {...білим кольором}
      Size:=White+1;
   end;
   SetAllPalette(pal);     {Змінюємо палітру: усі кольори білі}
   for k:=Blue to White do begin       {15 секторів, залиті...}
      SetColor(k);{...кольорами від Blue до White, але усі...}
      SetFillStyle(1,k);                    {...кольори - білі}
      PieSlice(GetMaxX div 2,GetMaxY div 2,k*24,(k+1)*24,50);
   end;
   repeat
     for c:=Blue to White do begin           {Ефект обертання}
        SetPalette(c,def_pal.Colors[c]);   {Відновлюємо колір}
        Delay(1000);
        SetPalette(c,EGAWhite);         {Знову заміняємо білим}
     end;
   until KeyPressed;
   CloseGraph
end.
```

В адаптерах VGA і IBM8514 є можливість при формуванні кожного з програмних кольорів здійснювати роздільне керування інтенсивністю кожного з основних кольорів (червоний, зелений, синій) за рахунок завдання їхніх часток у загальній інтенсивності формованого кольору. Для цього використовується процедура

SetRGBPalette(колір**,** червоний**,** зелений**,** синій**).**

Її перший параметр – номер програмного кольору (для VGA – від 0 до 15), що коректується; значення інших параметрів задають долі червоного, зеленого і синього кольорів у максимальній інтенсивності

кожного з них (значення від 0 до 63, які дають усього 2^{18} комбінацій кольорів).

```
{Приклад 11.23}
{Установлення VGA-палітри: за натисканням клавіш з літерами
 (R,r), (G,g), (B,b) здійснювати збільшення/зменшення
 часток червоного, синього, зеленого кольорів для всіх
 кольорів за винятком білого і чорного з відображенням процесу
 установленням кольору в зафарбованих прямокутниках}
uses Graph,Crt;
var
   d,m,x:Integer;
   k:Byte;
   r,g,b:Byte;              {Долі червоного, синього, зеленого}
   a:array[1..8] of Integer;
   ch:Char;
const
   p:array[Black..White] of ShortInt=            {EGA-палітра}
      (0,1,2,3,4,5,20,7,56,57,58,59,60,61,62,63);
begin
   d:=Detect;
   InitGraph(d,m,'c:\tp7\bgi');
   if GraphResult<>0 then WriteLn('Помилка');
   x:=50;
   for k:=Blue to Yellow do begin
      SetFillStyle(1,k);              {Стиль і колір заливання}
      a[1]:=30*k;a[3]:=(k+1)*30;a[5]:=a[3];a[7]:=a[1];
      a[2]:=100;a[4]:=100;a[6]:=150;a[8]:=150;
      FillPoly(4,a)          {Малюємо зафарбований прямокутник}
   end;
   for k:=Blue to Yellow do begin {Цикл установлення кольорів}
      r:=0;g:=0;b:=0;         {Частки кольорів дорівнюють нулю}
      repeat
         SetRGBPalette(p[k],r,g,b);      {Установка кольору k}
         ch:=ReadKey;
         case ch of   {Збільшення і зменшення часток кольорів}
            'R':if r<63 then Inc(r) else Write(#7);
            'r':if r>0 then Dec(r) else Write(#7);
            'G':if g<63 then Inc(g) else Write(#7);
            'g':if g>0 then Dec(g) else Write(#7);
            'B':if b<63 then Inc(b) else Write(#7);
            'b':if b>0 then Dec(b) else Write(#7);
         end;    {При натисканні <Enter> колір k установлений}
      until ch=#13;
   end;
   repeat until KeyPressed;
   CloseGraph
end.
```

12. МОДУЛЬ STRINGS

У TP немає процедур і функцій, призначених для роботи з ASCIIZ-рядками. Для цих цілей створений спеціальний модуль **Strings**, у якому зібрані функції (не процедури!), що забезпечують можливості оперування довгими рядками. При цьому повинен бути використаний розширений синтаксис (директива **{$X+}**), що, крім того, забезпечує можливість звертання до функцій як до процедур (а це особливо актуально в модулі **Strings**, у якому утримуються тільки функції).

12.1. Створення ASCIIZ-рядків у динамічній пам'яті

Розміщення довгих рядків у динамічній пам'яті провадиться функцією

StrNew(ASIIZ-рядок**)**,

що створює в динамічній пам'яті копію рядка, заданого як параметр (значення, що повертається, має тип **PChar**).

Для видалення ASCIIZ-рядка з динамічної пам'яті застосовується функція

StrDispose(ASCIIZ-рядок**)**,

у якої нема значення, що повертається, у зв'язку з чим до неї можна звернутися тільки як до процедури з обов'язковим включенням розширеного синтаксису.

```
{Приклад 12.1}
{Створення і знищення ASCIIZ-рядка в динамічній пам'яті}
{$X+}                          {Включення розширеного синтаксису}
uses Strings;
type
   MaxPCharType=array[0..65534] of Char;      {Максимальний...}
                                    {...розмір символьного масиву}
var
   ASCIIZ:PChar;
   p:^MaxPCharType;
begin
   New(p);           {Створення динамічного символьного масиву}
   WriteLn('Введи ASCCIZ-рядок');
   ReadLn(p^);                          {Введення даних у масив p^}
   ASCIIZ:=StrNew(p^);                  {Створення копії масиву p^}
   WriteLn(p^);
   Dispose(p);
```

```
   WriteLn(ASCIIZ);                {Змінна збереглася навіть...}
                                          {...після знищення p^}
   StrDispose(ASCIIZ);                {Знищення довгого рядка}
end.
```

Аналогом функції **Length** при обробці ASCIIZ-рядків є функція

StrLen(ASCIIZ-рядок**)**,

що повертає поточну довжину ASCIIZ-рядка без урахування завершального символу (тип **Word**).

Функція

StrEnd(ASCIIZ-рядок**)**,

що має тип **PChar**, повертає посилання на завершальний символ. Її можна використовувати для посимвольного перегляду ASCIIZ-рядків, а також для визначення їхніх довжин.

```
{Приклад 12.2}
{Чи правда, що ASCIIZ-рядок зліва направо і справа наліво
 читається однаково? При негативній відповіді переписати
 її "дзеркальне відображення" в інший ASCIIZ-рядок}
{$X+}
uses Strings;
type
   t=array[0..65534] of Char;
var
   ASCIIZ:^t;
   Mirror:PChar;
   i:Word;
   b:Boolean;
begin
   New(ASCIIZ);          {Створення динамічного символьного...}
                                    {... масиву з нульовою базою}
   WriteLn('Введи ASCCIZ-рядок');
   ReadLn(ASCIIZ^);
   WriteLn(ASCIIZ);
   WriteLn('Довжина рядка ',StrLen(ASCIIZ^));
   {Можна WriteLn('Довжина рядка ',StrEnd(ASCIIZ^)-ASCIIZ^);}
   b:=True;
   for i:=1 to (StrEnd(ASCIIZ^)-ASCIIZ^) div 2 do     {Цикл...}
           {...до середини довгого рядка (адресна арифметика)}
      if (ASCIIZ^+i-1)^<>(StrEnd(ASCIIZ^)-i)^ then begin
         b:=False;Break
      end;
   if b then WriteLn('Так')
   else begin
      WriteLn('Hi');
```

```
      Mirror:=StrNew(ASCIIZ^);         {Створюємо копію ASCIIZ}
      Mirror:=StrEnd(Mirror);  {Покажчик на заверш. символ #0}
      for i:=0 to StrEnd(ASCIIZ^)-ASCIIZ^-1 do begin
         Dec(Mirror);                  {Зміна значень покажчика}
         Mirror^:=ASCIIZ^[i];          {Копіювання символів;...}
      end;                    {...можна Mirror^:=(ASCIIZ^+i)^ }
      WriteLn(Mirror);
      StrDispose(Mirror);               {Знищення довгого рядка}
   end;
   Dispose(ASCIIZ);
end.
```

Функції

StrLower(ASCIIZ-рядок**)**

і

StrUpper(ASCIIZ-рядок**)**,

що мають тип **PChar**, перетворюють відповідно усі великі латинські літери в малі і навпаки, не здійснюючи перетворення інших символів (функція **UpCase** забезпечує перетворення тільки в одну бік і тільки одного символу).

```
{Приклад 12.3}
{Ввести ASCIIZ-рядок і вивести його на екран, задаючись
 регістром латинських літер у діалозі}
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
var
   ASCIIZ:^t;
   pt:^t;
   c:Char;
begin
   New(ASCIIZ);ReadLn(ASCIIZ^);
   WriteLn('U - великі, L - малі, інакше - без перетвор.');
   ReadLn(c);
   case c of
     'U','u':WriteLn(StrUpper(ASCIIZ^));
     'L','l':WriteLn(StrLower(ASCIIZ^));
     else WriteLn(ASCIIZ^)
   end;
   Dispose(ASCIIZ);
end.
```

12.2. Копіювання і конкатенація довгих рядків

Рядки типу **string** і довгі рядки можуть копіюватися один у одного. Для переходу від типу **PChar** до типу **string** служить функція

StrPas(ASCIIZ-рядок**)**,

що повертає значення типу **string**. Зворотне перетворення здійснює функція

StrPCopy(ASCIIZ-рядок, рядок**)**,

що переписує вміст другого параметра в довгий рядок, заданий як перший параметр. По закінченні роботи в імені функції (тип **PChar**) повертається покажчик на результуючий ASCIIZ-рядок.

```
{Приклад 12.4}
{Робота функцій StrPCopy і StrPas}
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
var
   ASCIIZ:^t;
   s:string;
begin
   New(ASCIIZ);
   s:='Turbo Pascal 7.0';
   StrPCopy(ASCIIZ^,Copy(s,7,6));        {Від string до ASCIIZ}
   WriteLn(ASCIIZ^);                  {Друкується рядок 'Pascal'}
   s:='Blez '+StrPas(ASCIIZ^);           {Від ASCIIZ до string}
   WriteLn(s);                   {Друкується рядок 'Blez Pascal'}
   Dispose(ASCIIZ);
end.
```

Аналогом функції **Copy** є функція

StrCopy(ASCIIZ-рядок1, ASCIIZ-рядок2**)**,

яка переписує другий рядок разом із завершальним символом #0 у перший і повертає покажчик (тип **PChar**) на початок першого рядка (на відміну від функції **Copy**, у якій результат виходить окремо). Функція

StrECopy(ASCIIZ-рядок1, ASCIIZ-рядок2**)**

виконує ті ж дії, що і **StrCopy**, але повертає покажчик на завершальний символ результату, що дозволяє організовувати з її допомогою конкатенацію рядків. В обох цих функціях другий

параметр може бути константою типу **string** чи виразом над такими константами.

```
{Приклад 12.5}
{Робота функцій SrCopy і  StrECopy}
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
var
   ASCIIZ1:t;
   ASCIIZ2,ASCIIZ3:PChar;
begin
   ASCIIZ2:=StrNew(ASCIIZ1);
   StrPCopy(ASCIIZ1,'Pascal');
   ASCIIZ2:=StrCopy(ASCIIZ2,ASCIIZ1);     {Звичайне копіювання}
   WriteLn(ASCIIZ2);                      {Друк рядка 'Pascal'}
   ASCIIZ3:=StrECopy(ASCIIZ1,'Turbo ');{У ASCIIZ1-'Turbo ';...}
        {...ASCIIZ3 посилається на завершальний символ ASCIIZ1}
   StrECopy(ASCIIZ3,ASCIIZ2);       {Копіювання в ASCIIZ3, ...}
                          {...рядок ASCIIZ1 нарощується справа}
   WriteLn(ASCIIZ1);                {Друк рядка 'Turbo Pascal'}
end.
```

У програмі після копіювання рядка 'Turbo ' у змінну ASCIIZ1 змінна ASCIIZ3 буде вказувати на завершальний символ рядка ASCIIZ1 за рахунок використання **StrECopy**. У результаті копіювання в ASCIIZ3 рядка 'Pascal' фактично відбувається подовження рядка ASCIIZ1. Вкладене звертання до функції **StrECopy** дозволяє замінити в програмі оператори

```
ASCIIZ3:=StrECopy(ASCIIZ1,'Turbo ');
StrECopy(ASCIIZ3,ASCIIZ2);
```

одним оператором

```
StrECopy(StrECopy(ASCIIZ1,'Turbo '),ASCIIZ2);
```

За допомогою функції **StrECopy** можна організувати багаторазове нарощування довгих рядків.

```
{Приклад 12.6}
{Функція введення з клавіатури ASCIIZ-рядка довільної
 довжини}
{$X+}
uses Strings,Crt;
function ReadLnASCIIZ(ASCIIZ:PChar):PChar;
var
   c:array[0..1] of Char;
   p:PChar;
begin
```

```
   c[1]:=#0;
   p:=ASCIIZ;                          {Покажчик на початок рядка}
   c[0]:=ReadKey;                                {Читання символу}
   while c[0]<>#13 do begin                      {Цикл до <Enter>}
      if c[0]=#0 then ReadKey  {Відсікання функц. клавіш і...}
      else if Ord(c[0])>31 then begin   {...керуючих символів}
         Write(c[0]);                    {Луна-відгук символу}
         p:=StrECopy(p,c);     {Копіюємо в p, покажчик змі-...}
      end;   {...щується на кінець рядка, ASCIIZ подовжується}
      c[0]:=ReadKey;                    {Читання нового символу}
   end;
   WriteLn;
   ReadLnASCIIZ:=ASCIIZ;   {Повертаємо покажчик на початок...}
end;                                      {...результуючого рядка}
type
   T=array[0..1000] of Char;
var
   ASCIIZ:T;
begin
   ASCIIZ:=New(T);
   WriteLn(ReadLnASCIIZ(ASCIIZ^));       {Друк покажчика на...}
end.                                     {...результат введення}
```

Функція

StrMove(ASCIIZ-копія, ASCIIZ-оригінал, кількість_символів**)**

копіює з другого рядка в перший рівно стільки символів, скільки зазначено в третьому параметрі (тип **Word**). Якщо довжина рядка, що копіюється, менше вказаного в третьому параметрі значення, то провадиться доповнення символами #0.

Функція

StrLCopy(ASCIIZ- копія, ASCIIZ- оригінал, max_кільк_символів**)**

працює аналогічно **StrMove**, але при довжині рядка, що копіюється, меншій значення третього параметра, доповнення символами #0 не провадиться.

В обох випадках можливий вихід за межі області пам'яті, відведеної під рядок-копію. В обох функціях значенням, що повертається, є покажчик на рядок-копію, а другим параметром може бути константа типу **string** чи вираз над такими константами.

```
{Приклад 12.7}
{Приклад на StrMove і StrLCopy}
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
```

```
var
   ASCIIZ:PChar;
   Pt:^t;
   c:Char;
   i:Integer;
begin
   New(Pt);
   for i:=0 to 999 do Pt^[i]:=#1;       {Заповнення рядка Pt^}
   Pt^[1000]:=#0;      {Занесення завершального символу рядка}
   ASCIIZ:=StrNew(Pt^);
   StrLCopy(ASCIIZ,StrPCopy(Pt^,'Turbo Pascal 7.0'),12);
   WriteLn(ASCIIZ);     {У ASCIIZ записано 'Turbo Pascal',...}
   WriteLn(Pt^);             {...а в Pt^ - 'Turbo Pascal 7.0'}
   StrMove(ASCIIZ,@ASCIIZ[6],StrLen(ASCIIZ));    {З ASCIIZ...}
        {...копіюються символи з позиції 6 (відлік від 0),...}
   WriteLn(ASCIIZ)     {...тобто рядок 'Pascal'; результат...}
   Dispose(Pt);StrDispose(ASCIIZ);   {...записується в ASCIIZ}
end.
```

Для зчеплення двох ASCIIZ-рядків у модулі **Strings** визначені дві функції:

StrCat(ASCIIZ-рядок1, ASCIIZ-рядок2**)**

і

StrLCat(ASCIIZ-рядок1, ASCIIZ-рядок2, максимальна_довжина**)**,

що повертають покажчик на результуючий ASCIIZ-рядок. Перша з них переписує другий рядок в область пам'яті, починаючи з байта, де розміщений завершальний символ першого рядка, а друга функція, виконуючи ті ж дії, обмежує результуючу довжину першого рядка значенням, яке задається третім параметром (тип **Word**). Другим параметром може бути константа типу **string** чи вираз над такими константами.

```
{Приклад 12.8}
{Конкатенація рядків за допомогою StrCat і StrLCat}
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
var
   ASCIIZ:PChar;
   Pt1,Pt2:^t;
begin
   New(Pt1); New(Pt2);
   WriteLn('Введи рядок');
   ReadLn(Pt1^);
```

```
   ASCIIZ:=StrCat(Pt1^,' - StrCat');   {Звичайна конкатенація}
   WriteLn(ASCIIZ);
   WriteLn('Введи рядок');
   ReadLn(Pt2^);
   ASCIIZ:=StrLCat(ASCIIZ,Pt2^,30);{Конкатенація з обмеженням}
   WriteLn(ASCIIZ);               {довжини результуючого рядка}
   Dispose(Pt1);Dispose(Pt2);
end.
```

Відзначимо, що в цій програмі Pt1^ і ASCIIZ служать для доступу до однієї ж тієї області пам'яті, будучи синонімами.

12.3. Пошук усередині довгих рядків

Функція

StrPos(ASCIIZ-рядок1, ASCIIZ-рядок2**)**

повертає покажчик на перше входження другого рядка в перший чи значення **nil**, якщо пошук виявився безуспішним. Другим параметром може бути константа типу **string** чи вираз над такими константами.

Для пошуку символу (у тому числі завершального символу #0) у довгому рядку можуть бути використані дві функції:

StrScan(ASZCIIZ-рядок, символ**)**

і

StrScan(ASZCIIZ-рядок, символ**)**,

що повертають покажчик відповідно на перше й останнє входження шуканого символу чи **nil**, якщо символ не знайдений.

```
{Приклад 12.9}
{Вивести на екран частину ASCIIZ-рядка:
 1) починаючи з позиції першого входження заданого підрядка;
 2) починаючи з позиції першого входження заданого символу;
 3) від початку по позицію останнього входження заданого
 символу}
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
var
   ASCIIZ:PChar;
   Pt1,Pt2:^t;
   s:string;
   c:Char;
begin
   New(Pt1); New(Pt2);
```

```
   WriteLn('Введіть ASCIIZ-рядок');
   ReadLn(Pt1^);
   WriteLn('Введіть рядок');
   ReadLn(s);
   ASCIIZ:=StrPos(Pt1^,StrPCopy(Pt2^,s));
   if ASCIIZ=nil then
      WriteLn('Входження підрядка не виявлене')
   else WriteLn(ASCIIZ);
   WriteLn('Введіть символ');
   ReadLn(c);
   if StrScan(Pt1^,c)=nil then      {Пошук першого входження}
      WriteLn('Входження символу ',c,' не виявлене')
   else WriteLn(StrScan(Pt1^,c));
   WriteLn('Введіть символ');
   ReadLn(c);
   if StrRScan(Pt1^,c)=nil then  {Пошук останнього входження}
      WriteLn('Входження символу ',c,' не виявлене')
   else WriteLn(StrLCopy(Pt2^,Pt1^,StrRScan(Pt1^,c)-Pt1^+1))
end.
```

12.4. Порівняння довгих рядків

При порівнянні ASCIIZ-рядків використовуються 4 функції, що повертають значення типу **Integer** (0, якщо рядки рівні; менше нуля, якщо перший рядок менше другого; більше нуля, якщо перший рядок більше другого). Порівняння провадиться до першого символу, яким відрізняються два рядка, чи до вичерпання одного з рядків.

Функції

StrComp(ASCIIZ-рядок1, ASCIIZ-рядок2**)**

і

StrIComp(ASCIIZ-рядок1, ASCIIZ-рядок2**)**

відповідно розрізняють і не розрізняють великі і малі латинські літери і розглядають порівнювані рядки цілком, а функції

StrLComp(ASCIIZ-рядок1, ASCIIZ-рядок2, максимальна_довжина**)**

і

StrLIComp(ASCIIZ-рядок1, ASCIIZ-рядок2, максимальна_довжина**)**

є аналогами відповідно функцій **StrComp** і **StrIComp**, але кількість порівнюваних символів не може перевищити значення, задане третім параметром (тип **Word**). Другим параметром у всіх цих функцій може бути константа типу **string** чи вираз над такими константами.

```
{Приклад 12.10}
{Дано два ASCIIZ-рядки. Який з них менший?
 Порівняти ці ж рядки за першими 5 символами}
```

```
{$X+}
uses Strings;
type
   t=array[0..1000] of Char;
var
   ASCIIZ:PChar;
   p:^t;
   i,k:Integer;
begin
   New(p);
   Writeln(Введи два ASCIIZ-рядки);ReadLn(p^);
   ASCIIZ:=StrNew(p^);
   ReadLn(p^);
   for i:=1 to 2 do begin
      if i=1 then begin
         k:=StrIComp(ASCIIZ,p^);
         WriteLn('При нерозрізненні великих і малих літер')
      end
      else begin
         k:=StrLComp(ASCIIZ,p^,5);
         WriteLn('При розрізненні великих і малих літер');
         Write('за першими 5 символами ')
      end;
         if k<0 then WriteLn('перший рядок менше')
         else if k>0 then WriteLn('перший рядок більше')
         else WriteLn('рядки рівні');
   end;
   StrDispose(ASCIIZ);
   Dispose(p);
end.
```

СПИСОК ЛІТЕРАТУРИ

1. Епанешников А.М., Епанешников В.А. Программирование в среде Turbo Pascal 7.0.– М.: Диалог-МИФИ. 1993.- 228 с.

2. Зуев Е.А. Язык программирования Turbo Pascal 6.0, 7.0.- М.: Радио и связь, Веста. 1993.– 384 с.

3. Марченко А.И., Марченко А.А. Программирование в среде Turbo Pascal 7.0.– М.: Бином Універсал; К.: ЮНИОР. 1997.- 496 с.

4. Сердюченко В.Я. Розробка алгоритмів та програмування мовою Turbo Pascal.– Х.: Паритет. 1995.- 352 с.

5. Турбо Паскаль 7.0.– К.: Издательская группа BHV. 1996.- 446 с.

6. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс. Учеб. пособие.– М.: Нолидж. 1997.- 616 с.

КОНТРОЛЬНІ ЗАПИТАННЯ

1.1. Що таке ім’я в ТР? Як підрозділяються імена?
1.2. Які стандартні скалярні типи існують в ТР?
1.3. Як організуються найпростіші введення і вивід в ТР?
1.4. Що таке константа в ТР?
1.5. Чим характерні відрізковий тип і перелічуваний тип?
1.6. Для чого служить розділ опису типів?
1.7. Які типи в ТР вважаються еквівалентними і сумісними?
1.8. Яким чином в ТР здійснюється перетворення типів?
1.9. Які бітові операції існують в ТР?
2.1. Які оператори застосовуються для організації розгалуження?
2.2. Коли доцільно застосовувати оператор **case**?
2.3. Які оператори застосовуються для організації циклів?
2.4. Чим відрізняються оператори **while** і **repeat**?
3.1. Що таке масив і як він описується в ТР?
3.2. Для чого служить директива компіляції **$R**?
3.3. Для чого використовуються змінні типу **string**?
3.4. Як можна довідатися фактичну довжину рядка?
3.5. Які оператори здійсненні над рядками?
3.6. Перелічити процедури і функції для роботи зі рядками.
3.7. Як оголошується множина в ТР?
3.8. Які дії можуть виконуватися над множинами в ТР?
3.9. У чому відмінність звичайних записів від варіантних?
3.10. Що таке текстовий файл і як він оголошується в ТР?
3.11. Перелічити процедури і функції для роботи з текстовими файлами.
3.12. У чому відмінність типізованих файлів від текстових?
3.13. Які засоби для роботи з типізованими файлами існують в ТР?
3.14. Яким чином можна обробляти помилки введення/виводу?
3.15. Для чого служать логічні пристрої?
3.16. Як можна здійснити перейменування і видалення файлів?
3.17. Як у ТР здійснюється створення і зміна каталога?
4.1. Що таке динамічна змінна і покажчик?
4.2. Які засоби існують у ТР для роботи з динамічними змінними?
4.3. Що таке динамічні списки і як вони організуються?
5.1. У чому відмінність процедур і функцій і для чого вони використовуються?
5.2. Як оголошуються і використовуються процедурні змінні?

6.1. Для чого служать модулі в TP?
6.2. Як оголошуються і підключаються модулі? Яка їх структура?
6.3. Які стандартні модулі існують в TP?
7.1. Для чого призначений модуль **Crt**?
7.2. Як здійснюється переключення текстових режимів?
7.3. Які засоби для роботи з екраном маються в модулі **Crt**?
7.4. Як забезпечується робота зі звуком у модулі **Crt**?
7.5. Для чого служать функції **ReadKey** і **KeyPressed**?
8.1. Що таке об'єкт і як він оголошується в TP?
8.2. Що таке інкапсуляція, спадкування і поліморфізм?
8.3. Як забезпечується спадкування і перевизначення?
8.4. У чому відмінність віртуальних і статичних методів?
8.5. Для чого служать конструктори і деструктори? Чи можуть вони бути віртуальними?
8.6. Чим відрізняються віртуальні методи від динамічних?
9.1. Як можна настроювати змінні за адресами?
9.2. Як можна обробляти командний рядок запуску програми?
9.3. Що таке безтипові параметри підпрограм?
9.4. Якими засобами можна здійснювати різні вставки в програму на TP?
10.1. Як запускаються зовнішні програми і команди DOS?
10.2. Як здійснюється робота з таймером у модулі **Dos**?
10.3. Які засоби для роботи з файлами і каталогами передбачені в модулі **Dos**?
10.4. Як обробляються переривання засобами модуля **Dos**?
10.5. Як перевизначити стандартну реакцію на переривання?
11.1. Як ініціалізується графіка в TP?
11.2. Як задаються координати точки?
11.3. Які процедури використовуються для побудови прямої?
11.4. Як задаються колір фону і графіки?
11.5. Як здійснюється зафарбовування областей?
11.6. Які процедури служать для побудови геометричних фігур?
11.7. Які процедури використовуються для виводу тексту в графіці?
11.8. Як організувати багатосторінкову роботу?
11.9. Як забезпечується зчитування і вивід спрайтів?
11.10. Що таке палітра? Якими засобами вона може бути запам'ятована і змінена?
12.1. Що таке довгі рядки і чим вони відрізняються від звичайних рядків?
12.2. Які процедури існують в модулі **Strings** для роботи з довгими рядками?

ЗАДАЧІ ЗА ГЛАВАМИ

До глави 2.

1. Дані дійсні числа *a, b, c.* Чи можна побудувати трикутник, довжини сторін якого дорівнюють цим значенням?
2. Дані додатні дійсні числа *a* і *r.* Чи можна описати коло радіуса *r* навколо рівностороннього трикутника зі стороною *a.*
3. Дано натуральне число *n.* Обчислити:

   а) $n!$; *б)* $\sqrt{3+\sqrt{6+\ldots+\sqrt{3(n-1)+\sqrt{3n}}}}$;

   в) $\left(1+\frac{1}{1^2}\right)\left(1+\frac{1}{2^2}\right)\ldots\left(1+\frac{1}{n^2}\right)$; *г)* $\frac{\cos 1}{\sin 1}+\frac{\cos 1+\cos 2}{\sin 1+\sin 2}+\ldots+\frac{\cos 1+\ldots+\cos n}{\sin 1+\ldots+\sin n}$;
4. Дані дійсне число *x* і натуральне число *n.* Обчислити:

   а) x^n; *б)* $x(x+1)\ldots(x+n-1)$; *в)* $x(x-n)(xa-2n)\ldots(x-n^2)$

   г) $\frac{1}{x^2}+\frac{1}{x^2}+\frac{1}{x^4}+\ldots+\frac{1}{x^{2^n}}$; *д)* $\frac{1}{x}+\frac{1}{x(x+1)}+\ldots+\frac{1}{x(x+1)\ldots(x+n)}$.
5. Дані дійсні числа *a, h*, натуральне число *n*. Обчислити
   $f(a)+2f(a+h)+2f(a+2h)+\ldots+2f(a+(n-1)h)+f(a+nh)$, де $f(x)=(x^2+1)\cos^2 x$.
6. Дано ціле число *n*. *а)* Скільки цифр у числі *n*? *б)* Чому дорівнює сума його цифр? *в)* Знайти першу цифру числа *n*. *г)* Дописати по одиниці в початок і кінець запису числа *n.*
7. Дано ціле число *n*. Знайти суму цифр числа *n* зі знакочергуванням (якщо запис *n* у десятковій системі є $\alpha_k\alpha_{k-1}\ldots\alpha_0$, то знайти $\alpha_k-\alpha_{k-1}+\ldots+(-1)^k\alpha_0$).
8. Дані натуральні числа *n* і *m.* Знайти: *а*) їх найбільший загальний дільник; *б)* їхнє найменше загальне кратне.
9. Нехай $\nu_1=\nu_2=0;\ \nu_3=1.5;\ \nu_i=\frac{i+1}{i^2+1}\nu_{i-1}-\nu_{i-2}\nu_{i-3},\ i=4,5,\ldots$ Дано натуральне число n ($n\geq 4$). Одержати ν_n.
10. Нехай $a_1=b_1=1;\ a_k=\frac{1}{2}(\sqrt{b_{k-1}}+\frac{1}{2}\sqrt{a_{k-1}});\ b_k=2a^2_{k-1}+b_{k-1},\ k=2,3,\ldots$ Дано натуральне число *n*. Знайти $\sum_{k=1}^{n}a_k b_k$.
11. Нехай $a_1=b_1=1;\ a_k=3b_{k-1}+2a_{k-1};\ b_k=2a_{k-1}+b_{k-1},\ k=2,3,\ldots$ Дано натуральне число *n*. Знайти $\sum_{k=1}^{n}\frac{2^k}{(1+a_k^2+b_k^2)k!}$.
12. Нехай $y_0=0;\ y_k=\frac{y_{k-1}+1}{y_{k-1}+2},\ k=1,2,\ldots$ Дано дійсне значення ε більше нуля. Знайти перший член y_n, для якого виконано співвідношення $y_n-y_{n-1}<\varepsilon$.
13. Знайти натуральне число *n*, що подається двома різними способами у вигляді суми кубів двох натуральних чисел $n=x^3+y^3$ ($x\leq y$).

14. Дано натуральне число *n*. Обчислити добуток перших *n* співмножників:
$$а)\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{5}{6}\cdot\ldots;\quad б)\frac{1}{1}\cdot\frac{3}{2}\cdot\frac{5}{3}\cdot\ldots$$
15. Дано натуральне число *n*. Одержати всі натуральні числа, що менші *n* і не мають з ним загальних дільників крім 1.
16. Дано натуральне число *n*. Одержати всі його прості дільники.
17. Дані натуральні числа *m* і *n* (*m*≤*n*). Знайти натуральне число між *m* і *n* з найбільшою сумою дільників.
18. Дано натуральне число *n*. Обчислити $\sum_{k=1}^{n}(-1)^{k+1}(3k^2-1)!$.
19. Дані дійсне число *x* і натуральне число *n*. Обчислити $\sum_{p=1}^{n}\sum_{q=p}^{n}\frac{x+p}{q}(p+q)!$.

До глави 3.

1. Дані дійсні числа $a_1,\ldots,a_n,b_1,\ldots,b_n$. Обчислити
$$(a_1+b_n)(a_2+b_{n-1})(a_n+b_1)\ .$$
2. Дано ціле число n. Скільки різних цифр у його десятковому записі?
3. Дані відмінні від нуля цілі числа $a_1,\ldots,a_n$. Якщо в послідовності числа з різними знаками чергуються, вивести вхідну послідовність; інакше вивести всі додатні члени послідовності, зберігши порядок їх проходження
4. Дана дійсна квадратна матриця порядку *n* (*n*≤15). Замінити в ній нулями всі елементи, що розташовані на головній діагоналі і вище її.
5. Дана дійсна матриця розміру *m*×*n* (*m*, *n*≤15). Знайти суму найменших значень елементів її стовпців.
6. Дана дійсна квадратна матриця порядку n (*n*≤15), натуральні числа *p*, *q* $(1\le p\le n, 1\le q\le n)$. Видалити з матриці *p*-й рядок та *q*-й стовпець.
7. Дано рядок *s*, що містить *n* символів $s_1,\ldots,s_n$. Якщо цей рядок є паліндромом, тобто $s_1=s_n$, $s_2=s_{n-1},\ldots$, то залишити його без зміни, інакше доповнити рядок справа його «дзеркальним» відображенням без повторення останнього символу ($s_1,s_2,\ldots,s_{n-1},s_n,s_{n-1},\ldots,s_2,s_1$).
8. Дано рядок. Видалити в ньому найдовший підрядок, що складається тільки з цифр.
9. Дані рядки символів *s1* і *s2*. Приєднати рядок *s2* справа до рядка *s1*. Якщо початок рядка *s2* збігається з кінцем рядка *s1*, при приєднанні рядків виключити таке накладення символів, видаливши найбільшу їх кількість. Наприклад, з рядків *абракадабра* і *брахман* повинен вийти рядок *абракадабрахман.*
10. Дані рядки символів *s1* і *s2*. Перелічити ті символи, що *а*) містяться в *s1*, але не містяться в *s2*; *б)* містяться й у *s1*, і в *s2*.
11. Дано рядок. Вивести всі його символи без повторення.

12. Існує інформація про 25 студентів, що містить прізвища, імена, по батькові і результати складання екзаменів з 5 дисциплін. Надрукувати прізвища тих студентів, які не одержали незадовільних оцінок і мають середній бал, що перевищує загальний середній бал.
13. Дано масив з 10 дат (число, місяць, рік). Вказати усі весняні дати, а також найпізнішу дату.
14. Дано текстовий файл *f*. Переписати в інший файл усі рядки вхідного файла, в яких відсутні латинські літери.
15. Дано текстовий файл. Переписати в інший файл частини рядків, починаючи з останнього слова, що не містить цифри.
16. Дано файл *f*, компоненти якого є дійсними числами. Знайти суму його додатних компонентів.
17. Дано файл цілих чисел. Записати в інший файл найбільше значення перших 10 компонентів, потім – наступних десяти компонентів і т. д. Якщо в останній групі менше 10 чисел, то розглядати неповну групу чисел.
18. Дано символьний файл *f*. Одержати файл *g*, утворений з файла *f* заміною всіх його великих латинських літер відповідними малими.
19. Дано файл цілих чисел *f* . Записати у файл *g* парні компоненти файла *f* у зворотному порядку.
20. Дано файл *f*, компоненти якого є цілими числами. Одержати файл *g*, утворений з файлу *f* виключенням повторних входжень того самого числа.

До глави 4.

1. Дано *n* цілих чисел, де *n* – натуральне число. Сформувати новий масив, що містить тільки ті елементи вхідного масиву, які є простими числами.
2. Дано матрицю $\boldsymbol{A}$ розміром $m \times n$, де m і n – натуральні числа. Одержати транспоновану матрицю $\boldsymbol{A}^T$, добутки $\boldsymbol{A}\boldsymbol{A}^T$, $\boldsymbol{A}^T\boldsymbol{A}$. Де сума елементів більше?
3. Дано числовий масив розміром $m \times n$, де m і n – натуральні числа. Не переміщаючи елементи масиву в пам’яті здійснити впорядкування рядків масиву за зменшенням суми їхніх елементів.
4. Дана квадратна матриця $\boldsymbol{A}$ розміром $n \times n$, де n – натуральне число. Обчислити її визначник. Використовувати рекурсивну підпрограму обчислення визначника розкладанням по першому рядку.
5. Ввести список академічної групи за абеткою і розмістити його у вигляді черги в динамічній пам’яті. У групу зарахований ще один студент. Помістити його прізвище в чергу, зберігши алфавітний порядок.
6. Розмістити в черзі *n* цілих чисел, де *n* – натуральне число, після чого перед кожним парним числом помістити значення, що дорівнюється половині цього числа.
7. Дано натуральне число *n*. Розмістити в динамічному стеці $2n$ дійсних чисел. Розглядаючи елементи стека відповідно до порядку обслуговування, здійснити таке перетворення списку: якщо число з парним порядковим номером менше попереднього в списку, змінити їх порядок слідування в списку, не змінюючи розташування в пам’яті.

До глави 5.

1. Дані цілі числа n_0, d_0, n_1, d_1, ..., n_7, d_7, a, b ($d_0d_1...d_7b\neq 0$). Обчислити $\frac{n_7}{d_7}\left(\frac{a}{b}\right)^7+\frac{n_6}{d_6}\left(\frac{a}{b}\right)^6+\ldots+\frac{n_0}{d_0}$, одержавши результат у вигляді простого дробу. Визначити підпрограми повного скорочення числа, заданого чисельником і знаменником, а також підпрограми множення і додавання простих дробів.
2. Дані додатні цілі числа n і m; обчислити функцію Акермана
$$A(n,m)=\begin{cases} m+1, & \text{якщо } n=0,\\ A(n-1,1), & \text{якщо } n\neq 0,\ m=0,\\ A(n-1,A(n,m-1)), & \text{якщо } n>0,\ m>0. \end{cases}$$
Використати рекурсивну підпрограму.
3. Скласти рекурсивну підпрограму, що обчислює добуток двох цілих чисел шляхом підсумовування: $xy=x+x(y-1)$, де $y>0$.
4. Оформити підпрограми обчислення площі трикутника за довжиною основи і висотою, площі прямокутника за довжинами його сторін, площі кругу за величиною його радіуса. У файлі записана інформація про тип фігур і їхні геометричні розміри. Користаючись процедурним типом, обчислити площі фігур, характеристики яких надані у файлі, і записати результат в інший файл.

До глави 6.

1. У модулі оформити підпрограми для перетворення в рядку великих (малих) українських літер у малі (великі), повороту вмісту рядка на 180°, перевірки того, що рядок є паліндромом і т. д. Використати ресурси модуля в програмі.
2. Оформити модуль для роботи з файлами цілих даних: підпрограми для пошуку позиції, в якій розташовується елемент даних, видалення у файлі елемента, що стоїть в заданій позиції, видалення у файлі всіх елементів із заданим значенням і т. д. Використати ресурси модуля в програмі.
3. Оформити модуль для обчислення значення визначеного інтеграла різними методами: метод прямокутників (з побудовою прямокутників за значенням функції на лівій, на правій межах і в середині інтервалу), метод трапецій, метод парабол. Використати ресурси модуля в програмі. Підінтегральну функцію задати у вигляді функції з трьома коефіцієнтами.

До глави 7.

1. У центрі екрана вивести текстовий рядок і забезпечити його переміщення вгору з відштовхуванням від нижньої і верхньої меж екрана. Передбачити примусову зміну напряму руху при натисканні клавіш зі стрілками <Вгору> і <Униз>. Вихід з програми – за натисканням клавіші <Esc>.
2. У центрі екрана у випадковому порядку на 1 секунду з'являються латинські літери і повідомлення <F1>, <F2>, …, <F12>, <Вгору>, <Униз>, <Вліво>, <Вправо>. Натискати на символьні і функціональні клавіші і клавіші

управління курсором відповідно сигналам, що надійшли. При натисканні клавіші <Esc> вивести дані про кількість правильно і неправильно натиснутих клавіш.

До глави 8.

1. Створити об'єктно-орієнтовану програму з двома рівнями спадкування: об'єкт верхнього рівня служить для обчислення визначника матриці третього порядку, а об'єкт-спадкоємець – для обчислення зворотної матриці для матриці третього порядку

До глави 10.

2. Написати підпрограму видалення файлів із вказаними розширеннями з заданого каталога. Обов'язкові параметри підпрограми – ім'я каталога і перелік розширень. Якщо файл, що видаляється, має атрибут **ReadOnly** чи **SysFile**, то перед видаленням здійснювати запит. Підкаталоги не видаляти.
3. Здійснити перейменування усіх файлів поточного каталога. Програма повинна видавати ім'я файла і запитувати нове ім'я і розширення. Відповіддю на запит повинні бути нові ім'я і розширення чи символ * замість імені чи розширення, якщо вони не змінюються. Підкаталоги не перейменовувати.
4. Здійснити перебір усіх exe- і com-файлів поточного каталога з видачею їх імен на екран. При виявленні потрібного файла запустити його на виконання з виходом з програми при першому запуску зовнішньої програми чи при вичерпанні exe- і com-файлів

До глави 11.

1. Реалізувати «гумову лінію». Відобразивши початкове положення графічного курсора в лівому верхньому куті екрана, переміщати графічний курсор за допомогою клавіш зі стрілками. При цьому точка останнього зафіксованого положення графічного курсора і точка його поточного положення повинні з'єднуватися прямою лінією. При натисканні клавіші <Enter> точка останнього положення графічного курсора фіксується як початкова, і подальша побудова лінії повинна здійснюватися стосовно неї. Вихід з програми – за натисканням клавіші <Esc>.
2. Зобразити на екрані секундомір. Початкове положення стрілки орієнтувати на 12 год. Запуск секундоміра здійснювати за натисканням клавіші <F1>, а припинення – за натисканням будь-якої клавіші. Повторне натискання клавіші <F1> повинно призводити до перезапуску секундоміра. При натисканні клавіші <Esc> програма припиняє роботу.
3. Дані 10 додатних чисел. Зобразити у вигляді вертикальних прямокутників різного кольору і стилю заливання діаграму, що відображає відсотковий вклад кожного з чисел у їх суму. Діаграма повинна бути масштабована на весь екран.

Навчальне видання

БЕЗМЕНОВ Микола Іванович

ТУРБО ПАСКАЛЬ 7.0

Навчальний посібник для студентів, які навчаються за напрямами «Прикладна математика» та «Інформатика»

Відповідальний за випуск *Л. Б. Кащеєв*

Роботу до видання рекомендував О. В. Горелий

Редактор В. М. Баранов

План 2003 р., п. 4/6

Підписано до друку 25.01.2006. Формат $60\times84^{1}/_{16}$.
Папір офсетний. Гарнітура Times New Roman. Друк офсетний.
Обл.-вид. арк. 14,6. Умов. друк. арк. 14,6. Наклад 400 прим. Зам. № 5.

Видавничий центр Національного технічного університету
«Харківський політехнічний інститут».
61002, Харків, вул. Фрунзе, 21.
Свідоцтво про державну реєстрацію ДК № 116 від 10.07.2002 р.

Парус™
Харків, вул. Щорса, 13 / 12.
Свідоцтво про видавничу діяльність ХК № 89 від 22.04.2003 р.

Надруковано у типографії ПП Стеценко І.І.